The DMI 2.0 Service Provider Software Development Kit Version 1.1

November 20, 1997
Version 1.1

October, 1997

Intel Corporation, 5200 NE Elam Young Parkway, Hillsboro, OR 97124-6497

Table of Contents

<This page is intentionally blank.>

Chapter 1 - Getting Started with the SDK

Overview

This DMI 2.0 Service Provider Software Development Kit (SDK) enables system developers to create and distribute the Desktop Management Interface (DMI) v2.0 technology on their platforms, integrate DMI Service Providers (DMI-SP) with their products, and integrate DMI-enabled management applications into their systems. The kit includes a DMI Service Provider and binaries that support the *DMI 2.0 Specification* for Microsoft Windows NT* and Windows* 95. All the sample source code included with this SDK was developed using Microsoft Visual C++* version 4.2.

See the **README.TXT** file for important setup information, errata for the *DMI 2.0 Specification*, and any limitations in this SDK.

Files included in this SDK

The SDK includes source files and libraries, code samples that illustrate how to build management applications and write component instrumentation, documentation detailing the SDK. The SDK also includes DMI development tools: DMI Explorer, DMI Component Test System, DMI MIF Conformance Checker, and DMI2SNMP Mapper (see "*DMI development tools"* for more information about these tools).

Only the DMI-SP executable is *required* for development using the SDK's libraries and source files.

Chapter 6 lists all the files included in this SDK.

DMI 2.0 Service Provider files for Windows* 95 and Windows* NT

The DMI 2.0 Service Provider (DMI-SP) included in this SDK runs on Windows NT and on Windows 95. Its dual interface provides all the Management Interface and Component Interface functions of both the *DMI 2.0 Specification* and the *DMI 1.1 Specification* for DMI-enabled management applications, MIFs and instrumentation. Chapters 6 and 7 describe the files required for running the DMI-SP.

DMI 2.0 SDK files, code samples and documentation

Software Development Kit files

This SDK contains all of the source files needed to develop DMI code using the dual interface described in the *DMI 2.0 Specification*. The DMI v2.0 dual interface of the DMI-SP is documented in Chapters 2 and 6. The procedural libraries are documented in Chapter 7.

Code samples

The code samples in this SDK show how to use the procedural API of the DMI to develop management applications and direct interface instrumentation. Chapter 5 fully describes these samples:

- Trusted Code
- Multi-Timer

Documentation files

The SDK includes the following document:

- **REFMAN.PDF**
  This Reference Manual, in Acrobat* format.

- **LICENSE.TXT**

  This file details Intel's licensing policies covering use of this SDK and its contents.

- **README.TXT**

  This file provides last-minute information about the product that could not be included in the manual, instructions for accessing the online documentation, instructions for getting technical support and tips for installing the SDK. This file also provides information on the known issues in the SDK.

- **AUTHRPC.HTM–Creating Secure Remote Access to DMI Information**

  This DMTF white paper describes how security is implemented in DMI-enabled Service Providers, components and management applications.

- **SECURE2.0.HTM–DMI 2.0 Security Token Proposal**
  This DMTF white paper describes a proposal for implementing security tokens for DMI 2.0.
- **REFERENCES.HTM–DMTF Reference and Association Proposal**

  This DMTF white paper describes a proposal for standardizing references and associations.

You can also refer to the Frequently Asked Questions (FAQ) list about this DMI SDK, available in the Developer area of Intel's Managed PC page on the World Wide Web at http://www.intel.com/managedpc/. The DMTF home page has the *DMI 2.0 Specification* (which contains a bibliography of related documents) and MIF definition samples (such as the DMTF-approved Standard Groups definitions) available for downloading at http://www.dmtf.org. Finally, also refer to the references listed in *Customer support contacts* later in this chapter.

DMI development tools

The DMI 2.0 Service Provider Software Development Kit includes several development tools, which assist you in developing management applications and writing component instrumentation. These tools, detailed in the following sections, are included:

- DMI Component Test System
- Intel DMI Explorer
- DMI2SNMP Mapper
- DMI 2.0 MIF Conformance Checker
- Service Provider Setup Utility

The DMI Component Test System

The DMI Component Test System (DCTS2) is a Windows-based tool which allows the user to exercise DMI components and their instrumentation by iteratively executing a sequence of single DMI commands.

The Intel DMI Explorer

The Intel DMI Explorer assists you in the development of management applications and in writing component instrumentation. The DMI Explorer enables you to view and modify DMI-compliant component information.

The DMI2SNMP Mapper

The DMI2SNMP Mapper allows SNMP-based management applications to access DMI information on DMI-instrumented computers. The Mapper has two primary components: the MIF2MIB Generator and the SNMP Agent Extension. The MIF2MIB Generator translates DMI attributes into MIB data. The SNMP Agent Extension maps SNMP management requests into DMI commands, and transforms DMI events into SNMP traps.

The DMI 2.0 MIF Conformance Checker

The DMI 2.0 MIF Conformance Checker (COMPCHK2) is a Windows-based tool which allows you to:

- Check candidate MIF files for syntactic correctness, with respect to the component MIF grammar within the *DMI Specification.*
- Compare candidate MIF group definitions against reference MIF groups definitions (specified by the user) to ensure that matching group definitions map exactly to each other. All candidate groups that appear to be Standard Groups definitions must be both matched with a reference group and map exactly to its definition. This ensures that candidate MIFs contain only definitions actually approved by DMTF Standard Groups.

- Match and compare candidate MIF groups against required groups expressed in REQ files (REQ files are composed and specified by the user). Note that the REQ files also specify properties required by specified groups and attributes.

The Intel Service Provider Setup Utility for OEMs

The Intel Service Provider Setup Utility for OEMs is an InstallShield* setup program, which silently installs the DMI 2.0 Service Provider (SP) and required components on a Windows 95 and Windows NT systems. This makes it easier for the OEM to integrate the DMI-SP installation program with their installation program.

- The OEM setup program does not have any user interface elements. All windows and dialog boxes during the setup process are invisible to the end user.
- OEMs may use this utility in conjunction with their own product setup program to install the Intel DMI 2.0 Service Provider on their customers' systems.

Using this manual

Purpose and scope

This *SDK Reference* details the DMI v2.0 support implemented in this SDK. It covers topics not covered in the *DMI 2.0 Specification* and includes examples and detailed implementation-specific information.

Viewing this *Reference* online

This *Reference* is available in Adobe Acrobat PDF format. Files with a PDF extension require the Acrobat Reader to be viewed. If you don't have the Acrobat Reader, download a free copy of the program from the World Wide Web at **http://www.adobe.com/acrobat**. Once the Acrobat Reader is installed, open ***%WIN32DMIPATH%*\DOC\REFMAN.PDF** to read the online *Reference*.

Use the arrow keys, the PAGEUP key and the PAGEDOWN key to move through the document. You can also go to specific pages or search for particular words or phrases. A table of contents and index are provided to help you find topics of interest. The Acrobat Reader also contains extensive online help; simply select the **HELP** menu in the Reader window.

Printing the online manual

To print a hard copy of this SDK Reference, choose Print from the Acrobat Reader **FILE** menu.

Typographical conventions

Designates:	Text formatting:
Titles of documents, references to chapter or section titles	*Italics*
Filenames, paths, reserved words, menu titles	ALL CAPITALS, sometimes in **boldface:** **D:\FILENAME.EXT**
Environment variables	***ALL CAPITALS***, *Italic:* ***%PATH%.*** For example, the *%WIN32DMIPATH%* environment variable designates the root directory used by both the SDK and the DMI 2.0 Service Provider included in it.

Referencing

DMI Specification, unless otherwise noted, refers to the *DMI 2.0 Specification* document. Unless explicitly noted, all references to chapters and sections found in this document refer to this *Reference.*

Installing the SDK

System requirements

This SDK requires an IBM-PC* or 100% compatible system with an Intel486™ processor (DX266 MHz) or faster, that uses either Windows 95 or Windows NT[1] as its operating system. (For optimal performance, a system with 16MB of RAM and an Intel 90 MHz Pentium® processor or faster is recommended.) If you install the SDK and all of its auxiliary files, the SDK will require up to 10 MB of disk space.

Note: You must log in with administrator privileges to add or remove the Service for DMI-SP on a Windows NT system. See *Windows NT related notes* in Chapter 6.

The Setup program

The Setup program for this SDK is run under Windows NT or Windows 95 and copies a set of binary files for DMI software development into your system. It also makes changes to the **\AUTOEXEC.BAT** file, the Windows registry and the Windows **SYSTEM.INI** file, if needed.

The Setup program adds a line in your **C:\AUTOEXEC.BAT** that sets the environment variable ***%WIN32DMIPATH%*** and copies the files into the ***%WIN32DMIPATH%*** directory tree. It also creates a new program folder and adds icons for those SDK options selected by the user. ***%WIN32DMIPATH%*** refers to the directory path in which you installed the SDK; the default path is C:\DMI\WIN32.

[1] The SDK and its DMI-SP support both versions 3.51 and 4.0 of Windows NT.

Preparation

You must remove any resident DMI-related programs before running the Setup program for this SDK. This includes the DMI 2.0 Service Provider and any direct interface programs. If these executables are not shut down prior to installing this SDK, the Setup program may report critical errors due to existing disk files being unavailable or already in use.

Pre-existing DMI installations

The Setup program replaces any pre-existing installations of the DMI 2.0 Service Provider files, **without performing backups**. It does not, however, delete any user files, disable any direct interface instrumentation, nor remove any DMI-enabled programs from the Windows Startup group.

Install options

You can use the Setup program to select which of the following SDK program elements to install:

- DMI 2.0 Service Provider files
- DMI 2.0 SDK files, code samples and documentation
- DMI development tools: DMI Component Test System and DMI 2.0 MIF Conformance Checker
- DMI Explorer
- DMI2SNMP Mapper

Note: You can install just the DMI Service Provider files to enable the DMI without providing any code for development. If you want to use the SDK, you must install both the DMI 2.0 Service Provider and the SDK files.

The DMI development tools included in this SDK are recommended, but not required, for DMI product development. You can also use any other DMI development tools available that support the DMI v2.0 technology.

Installation Directory

By default, the Setup program uses the C:\DMI\WIN32 directory. To change the Service Provider root directory, set the WIN32DMIPATH environment variable.

Registry Changes

The Setup program modifies the Windows NT and Windows 95 registry database. The configuration information is saved in the branch `HKEY_LOCAL_MACHINE`, under `SOFTWARE\Intel\DMI 2.0 SDK\Current Version`. The Setup program creates two branches: one for the Service Provider and one for the Client Front End API.

Service provider registry changes

The table below details the registry key structure for the Service Provider. The Service Provider registry information is placed under the branch `SOFTWARE\Intel\DMI 2.0 SDK\Current Version\Service Provider\Remoting`. This branch describes the Remote capabilities of Win32 Service Provider. If this branch is empty or missing, DMI-SP initializes DCE RPC with all available transports.

Registry Key/Name Value	Value Type	Post-installation Value	Description
`SOFTWARE\Intel\DMI 2.0 SDK\Current Version\Service Provider\Remoting\DCE RPC`			DCE RPC configuration.
`SOFTWARE\Intel\DMI 2.0 SDK\Current Version\Service Provider\Remoting\DCE RPC\Type`	`REG_SZ`	`dce`	RPC type.
`SOFTWARE\Intel\DMI 2.0 SDK\Current Version\Service Provider\Remoting\DCE RPC\Module`	`REG_SZ`	`WSDMIDCE.DLL`	DLL for DCE RPC.

`SOFTWARE\Intel\DMI 2.0 SDK\Current Version\Service Provider\Remoting\DCE RPC\Transports`	`REG_MULTI_SZ`	`ncacn_ip_tcp`
		`ncacn_spx`
		`ncalrpc`
			List of DCE RPC transports initialized by the DMI-SP for communication with remote management applications. You can specify any transport defined in the DMI 2.0 Specification that is also supported by DCE RPC.
			If the key is not set, the DMI-SP initializes all available transports.

Client front-end registry changes

The table below details the registry key structure for the Client Front End, which is placed under the branch `SOFTWARE\Intel\DMI 2.0 SDK\Current Version\Client\Remoting`. This branch describes the remote capabilities of Client Front End. If this branch is empty or missing, the Client Front End will try to load the local, DCE RPC, and RAP modules.

Registry Key/Name Value	Value Type	Post-installation Value	Description
`SOFTWARE\Intel\DMI 2.0 SDK\Current Version\Client\Remoting\Local`			Local interface configuration.
`SOFTWARE\Intel\DMI 2.0 SDK\Current Client\Remoting\Local\Type`	`REG_SZ`	`local`	Protocol type.
`SOFTWARE\Intel\DMI 2.0 SDK\Current Version\ Client\Remoting\Local\Module`	`REG_SZ`	`WDMI2API.DLL`	DLL for local interface.
`SOFTWARE\Intel\DMI 2.0 SDK\Current Version\ Client\Remoting\DCE RPC`			DCE RPC configuration.
`SOFTWARE\Intel\DMI 2.0 SDK\Current Version\ Client\Remoting\DCE RPC\Type`	`REG_SZ`	`dce`	Protocol or RPC type.

`SOFTWARE\Intel\DMI 2.0 SDK\Current Version\Client\Remoting\DCE RPC\Transports`	`REG_MULTI_SZ`	`ncacn_ip_tcp`
		`ncacn_spx`
		`ncalrpc`
			Transports over which the client receives indications when calling DmiIndicationListen. The client begins receiving indications by calling DmiIndicationListen or DmiIndicationListenExt. DmiIndicationListen uses this registry entry. DmiIndicationListenExt does *not* use this registry entry; it uses the transports explicitly specified to it. The empty value indicates that all available transports are initialized.

`SOFTWARE\Intel\DMI 2.0 SDK\Current Version\ Client\Remoting\DCE RPC\Module`	`REG_SZ`	`WCDMIDCE.DLL`	DLL for DCE RPC.
`SOFTWARE\Intel\DMI 2.0 SDK\Current Version\ Client\Remoting\DMI RAP`			RAP configuration.
`SOFTWARE\Intel\DMI 2.0 SDK\Current Version\ Client\Remoting\DMI RAP\Type`	`REG_SZ`	`rap`	Protocol type.
`SOFTWARE\Intel\DMI 2.0 SDK\Current Version\ Client\Remoting\DMI RAP\Module`	`REG_SZ`	`WCRAP.DLL`	DLL for RAP.

Other registry values

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\win32sl\shrink can also be set. Setting this key to a non-zero value activates the "shrink" option. When the "shrink" option is activated, all uninstalled components are removed from the database.

Setting this key has the same functionality as invoking the DMI-SP with the -r option.

Note: If HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\win32sl\shrink is set to 0, even if you specify the -r option, the "shrink" option is not activated.

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\EventLog\Application\win32slService\EventMessageFile contains the full path to the DLL with the Service Provider error messages for either entry into the NT EventLog or display in message boxes.

For NT event logging, the registry value HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\EventLog\Application\win32slService\TypesSupported must be set to the value 0x7.

MIF Files

The DMI-SP includes a MIF database file, called sldb.dmi, which contains the Service Provider database. It is placed in the C:\DMI\WIN32\MIFDB\ directory.

The Service Provider MIF, win32sl.mif, is placed in the C:\DMI\WIN32\MIFS\BACKUP directory.

Management applications treat the Service Provider as a component with ComponentID set to 1. As such, the DMI-SP must support the standard ComponentID group.

When the DMI-SP is loaded, it looks for the database file, sldb.dmi. If sldb.dmi does not exist, the Service Provider creates an empty database and installs the Service Provider MIF file, win32sl.mif.

Existing MIF Databases

After the updated files have been copied and the original system files altered, the Setup program checks for an existing DMI component MIF database.

Although this DMI-SP supports component MIFs written for DMI v1.x, it *does not* support MIF databases originally built by any DMI v1.x Service Providers. **The MIF database *must* be rebuilt the *first* time you use this DMI-SP.**

Completion

After the Setup program completes, you must reboot your system to enable the new DMI-SP. When the DMI-SP is invoked after being installed for the first time, it installs its own MIF file as **ComponentID 1**. You can install any other MIF files and instrumentation into the MIF database after this step has completed.

Invoking the Service Provider

On Windows 95, the Service Provider runs as a Windows application.

On Windows NT, the DMI-SP is an NT service, controllable via the NT control panel. When the Service Provider starts, it reports itself ready only when it can process DMI commands.

The DMI Service Provider is automatically invoked on system startup.

Invocation Parameters

You can specify these invocation parameters when invoking the DMI Service Provider:

`-c` Causes the Service Provider to run as a Windows application. (On Windows 95, the DMI-SP always runs as a Windows application).

`-i` Suppresses Service Provider's running icon.

`-m` Causes the Service Provider to re-install the service provider MIF file from C:\DMI\WIN32\MIFS\BACKUP.

-p Protects the Service Provider from being terminated after a user logs off, when the Service Provider is running as a Windows 95 service.

`-r` Causes the Service Provider to recreate the database without any of the uninstalled components that still remain.
When a component is uninstalled, it continues to take up space in the database. To physically remove the uninstalled components from the database, stop the win32sl service and restart it, specifying the -r option. After the database is recreated, the Service Provider will continue to function normally.
The database is only recreated if it actually contains uninstalled components.
Note: If `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\win32sl\shrink`, is set to 0, even if you specify the -r option, the database will not be recreated.

`-t` *`fileName`*
enables tracing DMI commands received into a file called *`fileName`*. *`fileName`* should include the full path and the directory separator should be \\ (not by \).

Invocation Errors

If the Service Provider encounters errors during startup, it reports a service-specific error, listed in the table below.

Problem Description	Error Code	Possible solution
Failed to allocate memory needed.	0x1	Free some memory.
Failed to set up the file or directory environment needed.	0x2	Check space and permissions on disk.
Failed to initialize the version string.	0x3	Possibly not enough memory.
Failed to create one of the Service Provider threads.	0x4	Possibly not enough memory
Failed to create the database file.	0x5	Check permissions on sldb.dmi.
Bad win32sl.mif file.	0x6	The win32sl.mif file has been corrupted. Replace it with the one in the installation package.
There was no database file and the win32sl.mif file does not exist in its expected location.	0x7	Copy win32sl.mif to its expected location.
The database file does not exist and could not be created.	0x8	Check permissions.
The win32sl.mif file could not be installed.	0x9	The win32sl.mif has been corrupted. Replace it with the win32sl.mif in the installation package.
Failed to copy the backup file.	0xa	The Service Provider tried to use the backup database file, but it could not be accessed. Check that the backup file exists and that the permissions are set correctly.
Database magic number error.	0xb	An invalid magic number was found in the database. Revert to the backup database or create a new one.
Old magic number found in database.	0xc	This is an old, unsupported version of the database file.
Database version error.	0xd	This is an old, unsupported version.
Failed to create one of the Service Provider threads.	0xe	Possibly not enough memory.
Invalid checksum in database file.	0x10	The database has been corrupted.
Failed to create one of the Service Provider threads.	0x11	Possibly not enough memory.
Failed to create one of the Service Provider threads.	0x12	Possibly not enough memory.
Failed to create one of the Service Provider threads.	0x13	Possibly not enough memory.

Customer support contacts

Intel's DMI support

Intel is one of the founding members of the Desktop Management Task Force (DMTF) and participates in various DMTF training activities and DMI developers conferences. In addition to creating and distributing DMI-SPs, SDK and development tools, Intel actively supports DMI technology in its own products through instrumentation and management applications, such as Intel LANDesk® tools.

Intel provides technical support for this SDK, this DMI-SP and the development tools included in the SDK installation. FAQs and Newsgroups are available for each product from the ***Developer*** support area of Intel's Managed PC home page on the World Wide Web.

Intel's Managed PC home page: **http://www.intel.com/managedpc/**
Email: **developer_support@intel.com**
Phone: **(800) 628-8686**

DMTF's DMI support

The DMTF provides documents, files and support through their site on the World Wide Web. As the DMI technology grows and develops, announcements and updates are distributed through these contact points. Free copies of the *DMI Specification* and updated reference MIFs that contain the latest revisions of the DMTF-approved Standard Groups definitions are available at these sites:

DMTF home page: **http://www.dmtf.org**
Internet FTP site: **ftp.dmtf.org**

Chapter 2 - Introduction to the Desktop Management Interface

Overview

The Desktop Management Task Force (DMTF) is a cooperative, industry-wide organization, formed in 1992 to develop and deliver specifications for building easy-to-manage PC systems. Catalyzed by customer demand, the DMTF charter is to define a common management framework for PCs, and support vendors in bringing management, ease of use, and control to networked and standalone PCs. Today, the DMTF has more than 145 active members. More than 220 products comply with the Desktop Management Interface *DMI 1.0 Specification*.

In response to the rapid growth and diversity of desktop systems and local area networks, DMTF member companies labored together to develop technology that would:

- Simplify the complexity of managing the desktop environment
- Offer a standard way to manage desktop systems
- Enable vendors to easily add manageability to their products
- Lower the total cost of ownership

The goal was to provide manageability for desktop systems through an interface that is:

- Independent of a specific computer or operating system
- Independent of a specific management protocol
- Independent of any specific processor or hardware platform
- Usable locally — no network required
- Easy for vendors to adopt
- Easily mapped to existing management protocols (for example, CMIP and SNMP)

The Desktop Management Interface allows desktop and server computers, hardware and software products, and peripherals—whether they are standalone systems or linked into networks—to be manageable and intelligent. It allows them to communicate their system resource requirements and to coexist in a manageable PC and server environment. For users, DMI offers a number of benefits: systems and software that are simple and easy to use, plug-and-play installation, and real-time system diagnostics and support. All of these benefits drive down the total cost of PC ownership by maximizing a user's control of the PC and minimizing the support burden.

The DMI addresses these needs directly by the creation of an abstraction layer between the management software and managed system components.

The original DMI version 1.0 was released in September, 1994 and used a data block interface. Various developers implemented DMI Service Providers—formerly called Service Layers (SL)—that were modeled on the DMI framework specified by the DMTF in versions 1.0 and 1.1 of the *DMI Specification*.

The power of DMI

The simplicity of the Desktop Management Interface provides powerful manageability for any product attached to or installed in a computer system.

- It provides management information in a consistent fashion across operating systems, environments, hardware platforms and architectures.
- It works regardless of what vendor provides the product to be managed, and regardless of what vendor provides the management application.
- It provides manageability to any desktop system or networked server.

This vendor independence and ability to work on all platforms means it is easy to develop new applications and services that manage desktop systems, resulting in a wider base of manageable products. Management applications no longer need to use their resources for translation and discovery schemes. Instead, they can use their resources to provide more and better management tools.

DMI allows different systems to work together, making interoperability among different desktop and server systems a real possibility.

It is easy for any product to become a manageable component because it is easy to completely describe any product in the Management Information Format (MIF). DMI-enabled components provide competitive levels of manageability because a wide range of information becomes available to management applications. The DMTF oversees and distributes standard MIF files for specific industry segments, and promotes its Standard Groups MIF definitions. This enhances component interoperability because management applications can rely on consistent information about manageable products that possess similar characteristics.

DMI technology can lower the cost of ownership for customers and for vendors. System management can become a proactive process instead of one which reacts to problems by troubleshooting and providing support. Management applications can deal intelligently with any errors or problems as they come up, instead of reacting to problems after they occur.

The DMI interface is designed to provide consistent information rather than replace existing network management protocols. The DMI can coexist with other protocols since a product's MIF data can easily be mapped to management protocols, such as SNMP, CMIP and other proprietary systems.

The DMI 2.0 structure

The Desktop Management Interface defines a set of properties that can be used within a single system (a standalone desktop, a node on a network, or a networked server).

The DMI property set consists of:

1. A format for describing management information,
2. A service provider that serves as an abstraction layer between components and management applications,

3. Two sets of APIs, one for service providers and components to interact, and the other for service providers and management applications to interact, and
4. A set of services for facilitating remote communication.

The DMI structure provides the following:

- The DMI Service Provider (DMI-SP) is a program that resides in the desktop system or server and is responsible for all DMI activities. This layer collects management information from products (whether system hardware, peripherals or software, stores that information in the DMI's database and passes it to management applications as requested.

  The DMI-SP must provide a **Management Interface** (MI) for management applications, and may optionally provide a **Component Interface** (CI) for manageable (instrumented) components. Both the MI and the CI definitions include a local block interface (level DMI v1.x) and a procedural interface.

  ⇒ The Management Interface (MI) is an API that provides the interface between the Service Layer and management applications and allows these applications to access, manage and control desktop systems and servers, components and peripherals. The MI offers a consistent interface for any management application to the various mechanisms used to obtain information from products and components within a desktop system or server. It also enables management applications to access information from any system or product regardless of vendor.

- The Component Interface (CI) is an API that handles communication between manageable elements and the DMI's Service Layer. The CI gives all hardware or software components (whether they're in or attached to a PC or server) a common method for describing their management attributes or features. The **management applications** are programs for interrogating, changing, controlling, tracking and listing the elements of a desktop or network server system. A management application **requests** to the DMI-SP, which returns **responses** for those requests to the management application. Usually, management applications receive **indications** generated by component instrumentation; however, they may also send indications about managed components to the DMI-SP.

  A management application can be a graphical user interface program, a network management agent, an installer program, a diagnostics program or a remote procedure call. Management applications rely on the DMI-SP to support the Management Interface, which controls communication between the Service Provider and management applications.

  The *DMI Specification* specifies that the MI must support:

  ⇒ A ***procedural interface*** for registered remoteable management applications (see *Remoteable DMI service users* later in this chapter).

⇒ A local ***block interface*** for backward-compatibility to DMI v1.x level management applications (within the limitations of either the operating system or the DMI-SP being used).

- The **manageable products**—also called **components—**are hardware or software products that reside within, or are attached externally to, a desktop computer or network server, and which can be modeled with meaningful attributes. A component's attributes can also be instrumented in a way that provides indications of runtime events. Management applications can adjust the operational characteristics of the component by setting its attribute values. Components that can be instrumented include word processors, spreadsheets, motherboards, operating systems, hard disks, CD-ROMs, graphics cards, printers, sound cards, and modems. Instrumented component providers are sometimes called **event generators**. DMI-enabled components can be shipped originally with the system or can be added later.

  Each manageable product provides a **Management Information Format** (MIF) **file** which defines the product's manageable **attributes** and classifies them by organizing them into **groups**.

  You can define attributes with static values, instrumented values, or both. Static attribute values change only upon the request of a management application or the DMI-SP managing it. Instrumented attributes can be changed by the component's own **instrumentation** code. Instrumentation is exercised directly when an application sends requests to change attributes in the DMI-SP's store of component MIF data, and indirectly by event generators when an indication that they send to the DMI-SP provokes a management application to change attribute values.

  Instrumentation code relies on the DMI-SP to support the CI, which controls the communication between the SP and manageable products. The *DMI Specification* specifies that the CI must support both a local procedural interface and a local block interface. Note that supporting the CI within the Service Provider itself is not required for the DMI-SP to comply with the *DMI Specification.*

- The **remoteable interface** layer of the DMI 2.0 is designed to provide remote access to DMI functionality while hiding the intricacies of manipulating the DMI v1.x data blocks.

  Remoteable management applications interact with the DMI-SP through the RPC mechanisms supported by the operating environment. Remoteable management applications are implemented with **RPC support layer** support, so that they can open remote sessions to any DMI-SP available across the network and use the procedural MI of the connected DMI-SP to manage its components (for more information, see *Remoteable DMI service users* later in this chapter). Management applications that use the local data block API (from DMI v1.x) can interact with the data block MI of the local DMI-SP only.

  The DMI-SP interacts with remoteable management applications through the RPC mechanisms supported by the operating environment. A DMI-SP contains an **RPC support layer** and will respond to remote procedure calls made by any registered

management application that has opened a remote session to its node. The DMI-SP also provides a local data block MI, for DMI v1.x backward compatibility purposes.

Standard RPCs provide remote Authentication and Privacy on the communications link. Appendix D of the *DMI Specification* provides a list of key references about related topics. Also see the white paper included in this SDK, *Creating Secure Remote Access to DMI Information.* To read the paper, click its icon in the DMI SDK program group or open the file **AUTHRPC.HTM** in the ***%WIN32DMIPATH%*\DOC** subdirectory.

Managing DMI information

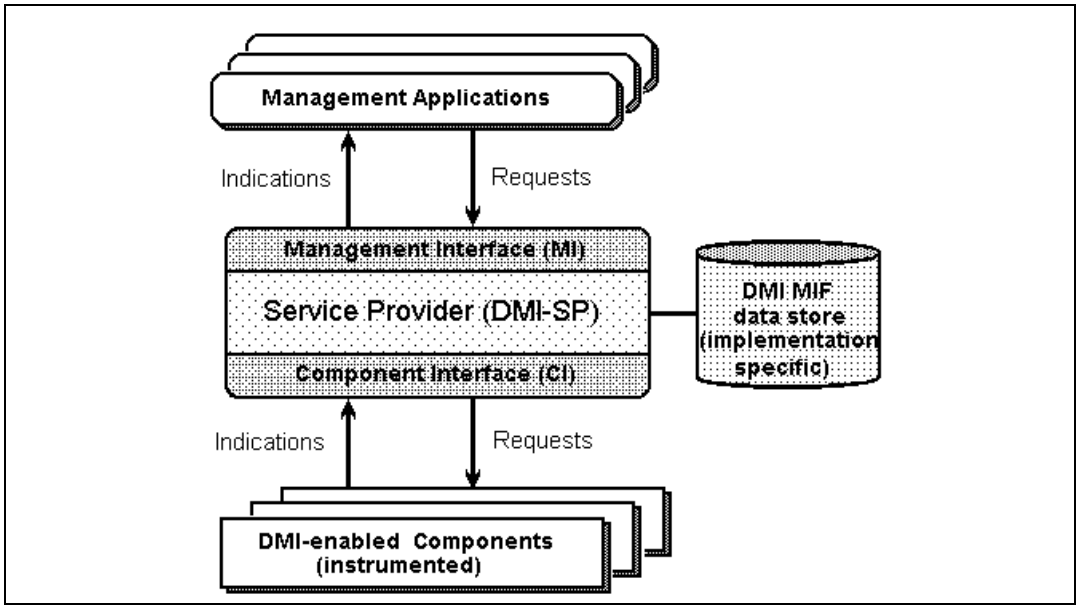
The DMI-SP is responsible for maintaining and delivering management information about all DMI-enabled components within its own local node. The general DMI data model describes a store of MIF data about managed components. The actual implementation used for the **MIF data store** or **MIF database** depends on whether the DMI-SP directly supports the CI described in the *DMI Specification* and on how the CI is implemented. Some DMI Service Providers use an internal database mechanism, while others use platform-specific APIs to access management data.

DMI-enabled management applications register with the MI of the DMI-SP to gain access to MIF data. The MI provides entry points to registered management applications for installing and removing component MIFs, accessing MIF data values and modifying the structure of installed component MIFs.

When the CI is present to support component instrumentation, valuable runtime information can be communicated to DMI entities, such as the Service Provider, manageable products and management applications, through the use of events and indications. The DMI-SP acts as the broker to communicate these management messages to those management entities which have registered to receive pertinent notifications.

The role of the DMI Service Provider

The DMI **Service Provider** (DMI-SP) is the manageability abstraction layer that manages all DMI activities. It is always ready for **requests** from **management applications,** and either **data requests** or **indications** from managed **components** (see the figure below).



DMI 2.0 functional diagram

The DMI-SP handles *Get*, *Set*, *List*, *Add*, *Delete* and implementation-specific functions called by management applications. The Service Provider retrieves requested information from the **component MIF data store** or **MIF database;** it performs maintenance on the MIF database and responds to the data requests made by manageable products. The DMI-SP also handles indications from manageable products and passes these notifications on to the management applications.

The *DMI Specification* specifies that the MI and the CI are **dual interfaces**; that is, each supports both a *procedural* interface and a *block* interface. The ***procedural MI and CI*** for the DMI 2.0 are detailed in the *DMI Specification*. The local ***data block MI and CI*** supported by the DMI 2.0 Service Provider are identical to those specified by the *DMI 1.1 Specification.*

The procedural MI includes an RPC support layer that coordinates remote sessions between the DMI-SP and remotely registered management applications. (See *Remoteable DMI service users* later in this chapter.)

Optional MI extensions

Chapter 9 of the *DMI Specification* contains a general description of functional extensions needed by the procedural MI—abstracted functions that are intended to isolate DMI service users from the intricacies of implementation-specific operations, such as coordinating memory functions and from using the RPC support layer of the DMI's remoteable interface. This is especially helpful when the management application developer wants to build an application to use DMI services through a particular RPC transport to communicate with DMI Service Providers (some of which may use different RPCs).

These MI support functions address and hide RPC specifics, such as:

- Establish and tear-down connections
- Present a unified error model to the client, hiding RPC-specifics details
- Handle memory allocation and release (to ease this burden for the user of the RPC mechanisms and to reduce the chance of introducing memory leaks)

Events and indications

Unsolicited messages sent between DMI entities through the **Component Interface** are called either events or indications. The terms are closely related and are sometimes used interchangeably. In this *Reference*, the term **event** describes the runtime condition being responded to by a component. The term **indication** refers to the notification message sent by an **event generator**[2] to alert the DMI-SP that the event occurred. The Service Provider then passes the **indication** to any DMI service users[3].

The DMI 2.0 defines an enhanced Event Model that filters and categorizes system indications, providing the management application with a higher level of intelligence about the change in a component's operating status. The Event Model allows indications from various system components to be sent to a management application to alert the user that there has been a change or there is a problem with the component.

Events generated by DMI components happen within the local node and are reported to the local DMI-SP. Indications, on the other hand, are sent by the local DMI-SP to any and all management applications that have registered to receive them. Since the DMI 2.0 supports a dual interface (both a local data block interface and a remoteable procedural interface routing through a layer of RPC support), the DMI-SP can report these localized events to registered DMI entities anywhere across the network. (See *Remoteable DMI service users* later in this chapter.)

Indications can be a powerful management tool. They can cover any runtime event, problem or error that a manufacturer decides needs attention. They can include a wide range of information about the event, problem or error, allowing intelligent, proactive response by management applications. Management applications look for indications so they can respond to error conditions. The data structures used in events and indications are described completely in the *DMI Specification*.

The role of management applications

Management applications present information about manageable products. Chapter 5 describes sample management applications, which illustrate how to retrieve management data from the DMI-SP and how to handle indications generated by instrumentation.

A management application may provide a diagnostic environment in which to exercise the instrumentation of managed components. Another kind of management application focuses on managing the data generated by a particular set of components. A

---

[2] Component instrumentation is one kind of an indication generator.

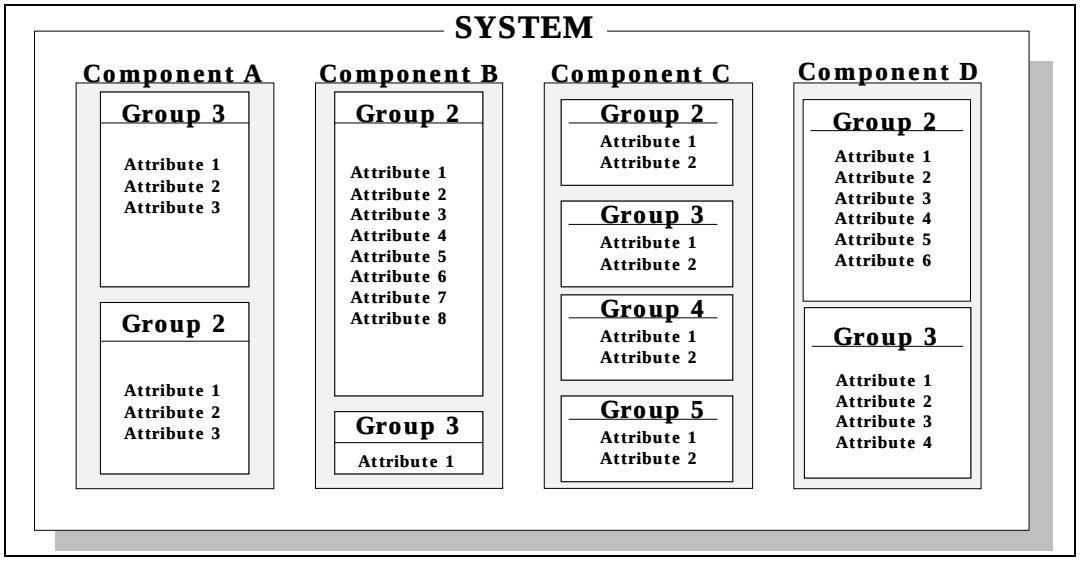
[3] Management applications that have registered to receive indications are an example of 'interested' DMI service users.

management application may present a browsing view of components, monitor the runtime behavior of selected components, or allow the user to perform management operations: component installation, removal, maintenance and upgrades. A management application can even be a subroutine or special control.

A DMI component provider is any product that provides management information to the DMI. Components provide their information in MIF files, which are installed into the component MIF data store. Attribute values may exist statically or be derived dynamically through component-specific instrumentation.

The component MIF data store

The DMI-SP builds, maintains and represents the management data of DMI-enabled components in an implementation-specific **component MIF data store**.[4] Products that have installed their MIF file into the system's DMI-SP are called DMI-enabled **components**.



The MIF data store model

The manageable attributes of a product are described with a language called the **Management Information Format**, or MIF. The MIF has a defined grammar and syntax. A **MIF file** is a simple ASCII text file describing a product's manageable **attributes**, classified into meaningful **groups** (see the "MIF data store model" figure above). The MIF syntax provides the means to use instrumentation code to provide up-to-date values to particular attributes within the installed component's MIF data.

[4] Because the DMI-SP provided with this SDK maintains the MIF data internally in a database format, this *Reference* will refer to this data store as either the 'component database' or 'the MIF database.'

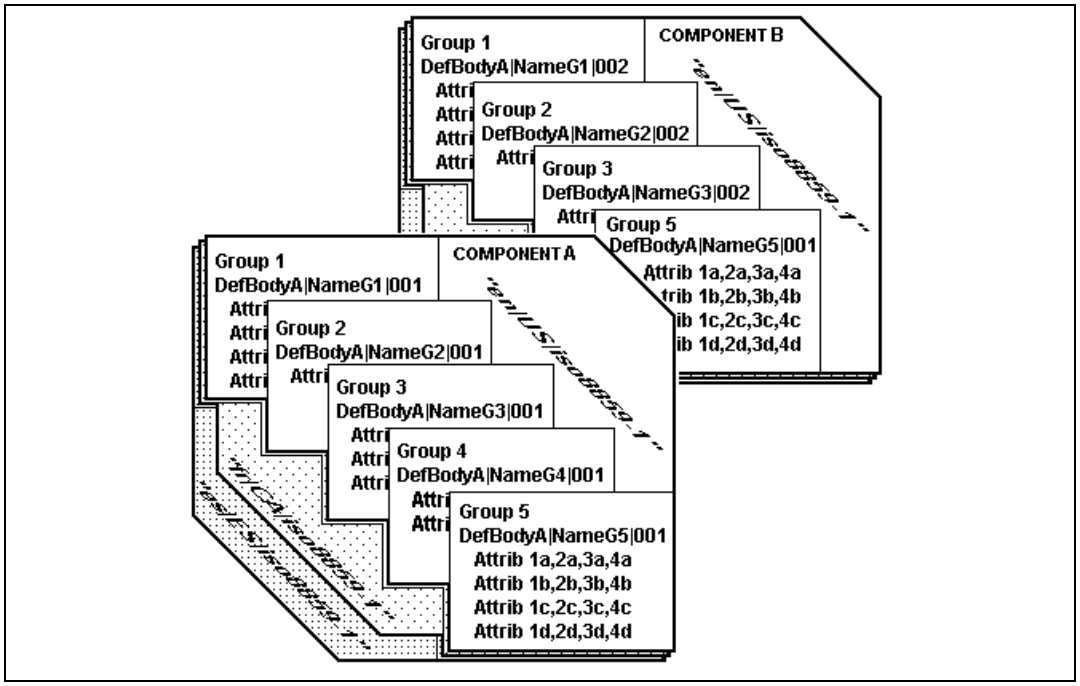
A group ID must be a non-zero, 32-bit unsigned integer and must be unique within the scope of the containing definition. As shown in the figure above, the groups found in a MIF do not have to appear in sequential order, nor is a Group 1 required; however, sequential ordering is more readable and easier to maintain.

Manageable products as component MIFs

Each product provides its own MIF file. When a manageable product is initially installed into the system, the information in its MIF file is added to the component MIF database and is then available to the Service Provider and thus to management applications.

Manageable information is defined as a set of **attributes,** which are organized into **groups** within the MIF data. The values associated with each MIF attribute are either stored directly in the component MIF database or provided by program code (instrumentation) that runs when the value is requested.

An attribute can be accessed individually, or as an entry in a **table** (with particular instances of that attribute within the table designated by their **keys**). Values can also be designated as *read-only*, *read-write* or *write-only*.



The MIF syntax data model

Each group definition includes a **class name** string. The generic formula for constructing a class name is as follows (refer to the "MIF syntax data model" figure above):

<Defining body>|<specific name of group>|<version number of this group definition>

For example, the class name string "DMTF|ComponentID|001" for the required Component ID group shows us that the DMTF defined the MIF encoding for this group; that this group defines the attributes about the identity of the component this MIF defines; and that this definition is the first formal version released for use with the DMI.

Different components can have similar attributes, such as peripheral devices that use a serial port, even though they may serve different needs (a serial printer versus an external faxmodem). If the attributes are grouped into well-focused categories, then dissimilar products that share a common set of attributes still can be categorized by class name for management applications that want to manage that class of attributes.

Because the DMI categorizes components by class name at the group level rather than the component level, we recommend that a MIF developer use DMTF-approved Standard Groups definitions to express product attributes whenever possible. These standardized group definitions were put together by designated members of the DMTF for just this purpose. Developers can augment their component MIFs with additional private groups for attributes that are unique or proprietary to their products.

The *DMI 2.0 Specification* has expanded the functionality in the MIF data model to include:

- Multiple language support for component MIF data
- Unicode support
- Multi-level support for pragma statements (at the component, group and attribute levels)

Language support

To facilitate the technology needs of the international marketplace, a component developer can create alternate language versions of the same component's MIF data. Installing these alternate copies of a component's MIF as overlays to a single component required changes to the *DMI Specification* itself. Therefore, all DMI 2.0 Service Providers support the component-level language functions, *DmiAddLanguage()* and *DmiDeleteLanguage()*, to enable a component's installation program to expand the languages defined for a single component's MIF data.

DMI language expressions are encoded in either Unicode 1.1 or ISO 8859-1 formats. Multiple language support enables the DMI user to:

- Define the language statement value of a single instance of a component's MIF data
- Set the current default language setting of the active DMI-SP
- Pass a value to MI functions to describe the preferred language for the data to be accessed

Unicode support

The MIF uses either the International Standards Organization document ISO 8859-1 (Latin Alphabet no. 1) or Unicode 1.1 specification for its character sets. If a Unicode MIF is provided, the first octet of the MIF file must be 0xFE (hexadecimal) and the second must be 0xFF. Otherwise, the DMI Service Provider will treat the file as an ISO8859-1 MIF.

Pragma support

Pragma definitions provide additional information about the component, group or attribute. As far as the DMI Service Provider is concerned, the literal value assigned to the pragma statement is simply an opaque octet string. Refer to the *DMI Specification* for a discussion of DMTF conventions for using pragma statements.

MIF definition tips

MIF file grammar, requirements and restrictions are described completely in the *DMI Specification*. Chapter 5 in this *Reference* gives some basic guidelines for writing effective MIF files. Additional documents about MIF definition and MIF samples are available from the DMTF home page. (See *Customer support contacts* in Chapter 1.)

Component instrumentation

A MIF file that contains only static data (data coded into the MIF file at the time of its creation) quickly becomes outdated, since the manageable aspects of most products change over time. A DMI component's **instrumentation** consists of code that maintains attributes with up-to-the-minute values and adjusts the component's operational characteristics based on these values.

By providing instrumentation, a component manufacturer provides accurate data to management applications, so that the management applications can make the best decisions for managing a system or a product.

Instrumentation[5] could be provided in one of the following ways:

- Direct interface programs, which are always running and registered with the Service Provider to provide the value on request for a specific group of attributes. Direct interface programs can provide persistent data, and are easily added to existing device drivers and other programs that must be loaded in the background.

- Runtime overlay programs, which are programs run by the Service Provider to retrieve or set the value at the time the action is requested by a management application. In Windows environments, runtime overlay programs are executed by the Service Provider as needed or they are provided as dynamic link libraries. Runtime overlay programs are useful for products for which no driver exists (such as the system BIOS or a serial port) or products where the information is transitory.

[5] Since instrumentation is platform-specific code, some types of instrumentation will not be supported by particular implementations of the DMI-SP Component Interface.

Note: Because runtime overlay programs are invoked by the Service Provider each time they are needed, they do not necessarily retain data between invocations. For persistent data, a direct interface program is more appropriate. This version of the Service Provider does not support runtime overlay programs.

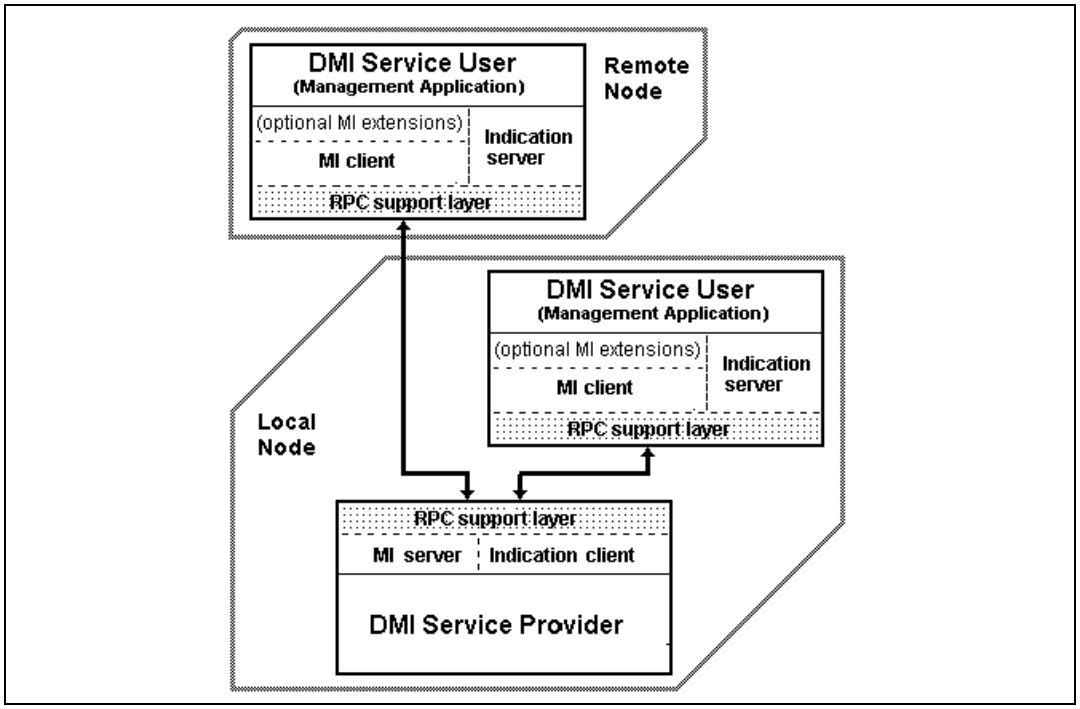
Remoteable DMI service users

The release of the DMI 2.0 further refines the DMI technology and extends the Service Provider's functionality to be usable by remoteable management applications. To enable the use of Remote Procedural Calls (RPCs) as the DMI's remoting agent, a procedural API specification has replaced the original data block API and indication support has been enhanced. Managing the DMI remotely is accomplished by a management application by first establishing a connection to that remote system, then registering with the remoteable DMI-SP in that system. The remoteable DMI-SP coordinates the flow of data and indications from the components it is managing to the remotely-registered management applications.

The DMI 2.0 Service Provider is designed to control communications between itself and remote management applications through a procedural Management Interface. This procedural MI includes an RPC support layer between the DMI entities to hide the complexity of coordinating management functions across a network. The DMI entities can concentrate on opening and closing remote sessions rather than other network-centric issues.

Connections between two RPC nodes can be made within the local system node entirely, or between a management application on a remote node and the local DMI-SP. (See the *Remoteable interface architecture* diagram later in this chapter. Note that this diagram shows only the remoteable MI interfaces.)

RPC is based on a client/server architecture. Communications between a ***DMI service provider*** (like the DMI-SP) and a ***DMI service user*** (like a management application) happen as client-server interactions coordinated by **RPC support layers** on either side.



Remoteable interface architecture

The initial connection is made when the DMI service user's RPC support layer subscribes to the DMI-SP's RPC support layer. Once connected, the associated client-server entities communicate with each other by procedure calls transparently across the RPC transport.

For performing management functions over the component MIF data, the DMI-SP supplies the **Management Interface server,** and the management application supplies the **Management Interface client**. For DMI implementations that support indications, the DMI-SP supplies the **Indication client,** and the management application supplies the **Indication server**.

The DMI 2.0 implementation in this SDK

The DMI Service Provider provided with this SDK is an implementation-specific entity. There are different implementations of the DMI-SP available for various operating systems and environments. To be considered DMI-compliant, a DMI-SP must support the functionality described in the relevant *DMI Specification.* However, different Service Provider implementations may have limitations or variations based on platform-specific requirements.

This SDK provides a DMI-SP that operates on only Windows 95 and Windows NT operating systems. The Management Interface of the DMI-SP in this SDK supports *only* 32-bit management applications, and the Component Interface supports all existing MIF definitions *except* that its instrumentation support is limited to *only* 32-bit direct interface instrumentation. Chapter 6 describes specifics of the architecture and implementation of the binaries provided with this SDK.

<This page is intentionally blank.>

Chapter 3 - Developing for DMI-enabled systems

Overview

The DMI Service Provider (DMI-SP) provides a common interface for use by the manageable products in a desktop or networked system, and by the management applications which need to access, control and communicate with those products.

A manageable product, also called a **component**, can be any piece of hardware or software installed in or connected to a desktop computer system. A DMI-enabled component is one that comes with a DMI MIF file, which describes its manageable characteristics, and program code (called instrumentation), which provides the data for those characteristics.

The DMI-SP interfaces with manageable products to maintain management information about them—either by storing the information in a database or by storing the pointers to their instrumentation. The DMI-SP can also collect information about exceptions, problems or errors from each component. These event notifications are called **indications** and are passed from components to the DMI-SP. The DMI-SP then passes this indication to any management applications that have registered to receive the notification. Indications provide powerful manageability for products that take advantage of them.

The Service Provider functions are available to manageable components and management applications regardless of which CPU, network, operating system, or environment is used. There are separate implementations of the Service Provider for each operating system environment. Each environment may have implementation-specific limitations.

This chapter describes how to enable management applications and components so they can work with the Desktop Management Interface and the Service Provider to provide manageable desktop and networked server systems.

Enabling management applications for DMI

Management applications can use the DMI *Get*, *Set*, and *List* functions to request information about manageable products in a DMI-enabled system. Each management application that wants to manage components on a particular system must register with the Service Provider for that particular system by calling a DMI *Registration* function.[6]

A management application can register to submit requests, receive notification of indications, or both. Management applications must provide code to deal appropriately with the

[6] The *Registration* functions depend on implementation-specific features of the operating system platform. Consult the documentation provided for the DMI-SP implementation to see how the *Registration* functions are supported by that DMI-SP.

status codes returned by the Service Provider. All status codes are listed in either the *DMI Specification* or the *%DMIDIR%*\WIN32\INCLUDE\DMI2ERR.H file included with this SDK.

A management application can also use the DMI *Add* and *Delete* functions to maintain the MIF data for particular components in the DMI-SP MIF database. These functions allow the application to add or delete component definitions (MIFs) and languages, add or delete group definitions, and add or delete rows to keyed groups.

The *DMI Specification* details the *Registration, Get, Set, List, Add* and *Delete* functions available to management applications. It also lists the status codes for these functions.

This SDK ships with sample management applications, which are described in Chapter 5.

Enabling manageable products for DMI

Follow these steps to enable a product for DMI.

To get started, you need:

- The *DMI 2.0 Specification*
  (available electronically from the DMTF on the World Wide Web, at www.dmtf.org).
- This *DMI 2.0 Service Provider SDK Reference.*
- The DMI Service Provider supported by your operating system
  (that also conforms to the *DMI 2.0 Specification*).

To produce a product that can communicate with the Service Provider as defined by the *DMI Specification*:

1. Read this *Reference* and the *DMI Specification* to understand how a component interfaces with the Service Provider and what functions must be supported.

2. Define what is manageable about your product and what specific features you want to showcase. Refer to existing standards to find MIF groups that may apply to your product. Sample MIFs are available from the DMTF on the World Wide Web. For more information about how to model the component and make good design decisions when writing a MIF file, see Chapter 4.

3. Begin with the Component ID group. Then define MIF groups based on a sensible breakdown of your product's features. Use as many of the existing Standard Groups that apply to your product as possible (Standard Groups are available from the DMTF). If needed, add proprietary groups for unique features of the product.

4. Define MIF attributes for each group. Decide whether the attributes are *read-only*, *read-write*, or *write-only*. Decide what instrumentation is needed to supply the data for each attribute value. Many attributes have values that change over time, and can be instrumented to be effective. Attribute values hard-coded into the MIF file can be unreliable or difficult to keep up to date. See the Manageable attributes and Instrumentation code sections later in this chapter for more detail about these decisions.

5. Create your MIF file, according to the grammar defined in the *DMI Specification*. (See the sample MIF definitions described in the *Specification.*) Use available development tools to check your MIF file for syntax errors and compliance to existing MIF standards. (See Chapter 8, *DMI 2.0 development tools*.)

6. Write instrumentation code for your component. Decide whether you will use the direct interface or runtime overlays[7] to provide information to the Service Provider about the component. (See Chapter 5, *DMI 2.0 code samples*.)

7. Modify your component product's install and uninstall programs so they properly install and uninstall your component's MIF file into the existing MIF database. See the sections *Installing your component* and *Uninstalling your component* in this chapter for more information about these requirements. For more details about the libraries that provide these installation procedures, see Chapter 7. Examples of installing a component's MIF file into the MIF database are included with the SDK and described in Chapter 5.

8. Decide what errors, exceptions or problems to flag to the Service Provider as events, and what information to include in each. Add a group to your MIF file describing each event, and code your instrumentation to notify the DMI-SP about the event. See the *DMI Specification* for information about the entry points used to pass events to the Service Provider.

   ⇒ These unsolicited messages form the basis of the indications to be sent by the component to the Service Provider. Indications make sophisticated, intelligent management possible in real time, because management applications are notified immediately when there is a problem.

Manageable attributes

Once you've decided what aspects of your product are manageable attributes and you've grouped these manageable attributes sensibly (and that use Standard Groups as much as possible), then you must define these attributes in the MIF syntax. You can find full descriptions of MIF attribute characteristics and options in the *DMI Specification*. All attributes must have the following four items defined:

- **Name:** The name of the attribute, such as "Baud rate" or "BIOS version number."
- **ID**: The AttributeID number, which must be unique within the group containing the attribute and sequential within the group.
- **Type:** The attribute's data type, either an enumerated list or a specific data type from those supported by the interface. An enumerated list provides a great deal of flexibility in attribute definitions.
- **Value**:[8] The attribute's actual value or a pointer to the value. Instrumented values (that is, attributes whose values are derived by de-referencing pointers to their instru-

[7] This SDK supports direct interface instrumentation only.

[8] There is one circumstance when the Value statement is not required in an attribute definition: that is when the attribute's group is keyed. The convention in this case is to

mentation) are most effective. Static values are best used for attributes that rarely change.

⇒ A value can also be an **enumeration** value, indexing into a table of values previously defined in the MIF file. Enumeration values can be *read-only* or *read-write*, but cannot be *write-only*.

⇒ Many attributes have values that change over time, and can be instrumented to be effective. For **instrumentation,** the MIF file provides a path to a runtime program or specifies a direct interface to the Service Provider. The product manufacturer is responsible for providing any instrumentation code. Each column in a table of data must get its data (including key value) either entirely from the database, or entirely by instrumentation. You cannot mix the two methods in one table column.

In addition, the Component ID group's attributes can optionally include:

- **Description:** A single character string which can occupy one or more lines of the MIF data. Descriptions are more useful for management applications when they supply clear, unambiguous, complete information about the attribute. Provide good Description lines at the component and group levels, as well.)
- **Access:** A keyword value that determines whether the attribute's value can be read or written. The options are *read-only*, *read-write*, or *write-only*; when not specified, the default value is *read-only*.
- **Storage:** A keyword value that provides a hint to management applications to assist in optimizing storage requirements. The options are *common* or *specific*; when not specified, the default value is *specific*.

  ⇒ Use *specific* to signify that there may be a large number of different values. An example of a specific attribute would be a component's serial number. Specific attributes are probably not good candidates for optimization.

  ⇒ Use *common* to signify that the value of this attribute is typically limited to a small set of possibilities. An example of a common attribute would be the clock speed of a microprocessor.

Instrumentation code

Instrumentation code is a program or API that is either already resident and connected to the Service Provider (direct interface instrumentation) or invoked by the Service Provider at the time its attribute values are requested (runtime overlay instrumentation).

Direct interface programs are useful for products using device drivers, or background programs and products for which persistent data is required.

---

define the Values for all attributes in that group in a separate Table definition block. When one of a keyed group's attributes has a Value statement defined, its value expression is used as a default value whenever a row's definition neglects to specify a non-NULL value for that row's instance of that attribute.

Runtime overlay programs are useful for products that provide transitory data, or products which are not associated with a device driver loaded in RAM or with a background task, such as the system BIOS. This SDK implementation does not support runtime overlay instrumentation. (See *Runtime overlay instrumentation* later in this chapter.)

This SDK provides several procedures that make writing instrumentation easy, and several examples of instrumentation code that use these procedures. The DMI procedure library provides procedures for installing a component, invoking the Service Provider and accessing attributes; they are described in Chapter 7. Implementation-specific issues for instrumentation code are included in Chapter 6. The rest of this Instrumentation code section summarizes the general issues and requirements for instrumentation code.

Component Providers

Instrumentation code must provide only one value at a time when invoked by the Service Provider. To be accessed by the Service Provider, all instrumentation code using the procedural interface must support the following DMI functions as entry points:

- CiGetAttribute
- CiGetNextAttribute
- CiSetAttribute
- CiReserveAttribute
- CiReleaseAttribute
- CiAddRow
- CiDeleteRow

The Service Provider supports the following DMI functions as entry points to be accessed by Component Providers:

- DmiRegisterCi
- DmiUnregisterCi

Use the code sample programs as the basis for your instrumentation code. See Chapter 5 for more information about code samples in this SDK.

Event Generators and Indication Consumers

Event Generators, like component instrumentation code, must utilize the Service Provider entry point, *DmiOriginateEvent()* function. Indication consumers, like management applications, should provide functions for Event Delivery; for more detail, see Section 7.1 in the *DMI Specification*.

Instrumentation development

There are two methods of instrumenting component MIFs through the DMI: direct interface and runtime overlays.[9]

Direct interface instrumentation

To use the direct interface, your component must register with the Service Provider as a direct interface. If you specify the keyword DIRECT-INTERFACE in the Path statement of

[9] The development libraries in this SDK *do not* support the development of runtime instrumentation, nor does this DMI-SP support existing runtime overlay instrumentation.

the MIF file, the Service Provider can return an appropriate error when accessing attributes when the direct interface program is not running.

Direct interface components must register by calling the Component Interface's *Register* function before doing anything else. In registering, the component passes its entry point to the Service Provider. The Service Provider passes control to the direct interface program when it needs to access the attribute value. When it unloads, a direct interface program must unregister with the Service Provider. The instrumentation code samples included with this SDK (Trusted Code and Multi-Timer) demonstrate how to do this.

Before installing your component into the Service Provider's MIF database and registering it as direct interface, confirm that the component has not already been installed or registered with the Service Provider.

To identify which ComponentID in the current MIF database belongs to your component, use the DmiListComponentsByClass() function to filter for the component class string that matches your component. To verify whether or not your component is registered with the Service Provider as direct interface instrumentation, perform a DmiGetAttribute() request. If the component is registered, the correct value will be returned. Otherwise, the Service Provider will return the "Direct Interface Not Registered" error code. These methods are demonstrated in the instrumentation code samples described in Chapter 5.

Runtime overlay instrumentation

The Service Provider in this SDK does not support runtime instrumentation, nor does its DMI-SP support the use of existing runtime overlay instrumentation. Discussion of runtime instrumentation is beyond the scope of this SDK Reference. See the *DMI Specification* for more information on this topic.

Installing your component

When the user installs your product into their system, your installation process is responsible for incorporating the component MIF file into the existing MIF database (or putting it into the MIF subdirectory, if the Service Provider is not running). When designing your installation process to accommodate the Desktop Management Interface, you must provide the MIF file (a text file that complies with the grammar found in the *DMI Specification*).

The DMI procedure library, WCDMI.DLL, includes the database administration functions *DmiAddComponent() and DmiAddLanguage()* for your setup program to use to add your component's MIF file(s) to the MIF database. These procedures also determine whether the Service Provider is running. This library is included with this SDK and is documented in Chapter 7.

This SDK includes some examples of installing your component's MIF into the MIF database; these code samples are explained in Chapter 5.

Uninstalling your component

When providing a DMI-enabled component, we highly recommend that your product include an uninstall program, as well as an install program. The uninstall process should

make sure that any direct interface component instrumentation currently running and registered calls the *DmiUnregisterCi()* function. Then the uninstall process should remove the component from the MIF database by calling the *DmiDeleteComponent()* function. This removes incorrect component information from the MIF database.

To call the function, your uninstall program must first identify which ComponentID in the current MIF database belongs to your component. You can do this with several methods, such as using the *DmiListComponentsByClass()* to filter for the component class string that matches your component. This method is demonstrated in the instrumentation code samples described in Chapter 5.

Other resources for developers

The *DMI Specification* provides a full technical description of the Desktop Management Interface (its data definitions, functions and options) and the MIF grammar and syntax.

This *Reference* addresses many of the development tasks necessary to enable products and systems for the DMI. Chapter 4 provides guidance for modeling your product in a way that makes sense and for constructing a good MIF file. Chapter 5 explains the DMI 2.0 sample code included with the SDK. Chapter 6 describes the specific Service Provider implementation included in this SDK. Chapter 7 describes the SDK's procedure libraries.

<This page is intentionally blank.>

Chapter 4 - Modeling products with MIF files

Overview

A MIF file is a text file[10] which follows the syntax and grammar set out in the *DMI Specification.* A MIF file can contain as few as one group with the four standard attributes or as many groups and attributes as you choose. The real value of a MIF file is that it codifies how you model your product and how you think your product should be managed in the desktop system.

The DMTF recognizes that desktop systems are dynamic. Therefore, the DMI does not assume that a management application can know anything about the components on the desktop. It is up to the component to provide all meaningful information by means of the MIF file. Consequently, it is crucial to design the MIF file well.

This chapter discusses modeling methods and the basics of good MIF design. This SDK includes a tool for checking MIF syntax, to help debug the MIF files you write.

Modeling a product for management

Components are products. Modeling a component is the process of determining what characteristics of the product can be managed, then grouping those characteristics in ways that make sense. Good product models also allow for expansion and change, since every product will change over time—adding new functions, options and capabilities.

The *DMI Specification* is written so that classification occurs at the group level, rather than at the component level. This allows much flexibility for conforming to the standard. The greatest effort in writing a MIF file should go into determining the groups and the attributes contained in those groups.

Setting out the product's attributes

First, detail all the attributes of your product that can be managed. Think of all the product elements that are configurable or change in the course of operation. Consider looking at some existing MIF files to get ideas for the kinds of attributes that make a product manageable.

Defining groups to classify attributes

After you detail all the attributes for your product, you must group the attributes sensibly. Groups of attributes should be conceptually distinct or strongly related. Ideally, they should relate to a physical thing, especially for hardware products.

---

[10] The text may be in either ISO 8859-1 format (editable in a standard ASCII editor) or in Unicode 1.1 format (requires a Unicode editor).

For instance, a manufacturer who produces a motherboard would consider the motherboard to be the component. There are groups to define attributes for the processor, the serial port, and the parallel port, but not separate groups for the memory management unit or the write-through cache, which are integral to the processor itself.

For software products, the groups should map to functionally-related items. For example, a word processing product might have groups to describe the editing function, the spell-checker, the drawing tools and the printing function.

Consider how the product will be used when you model it; choose groups and attributes that will make sense to the people managing it. Use standardized groups and attributes as much as possible, because these definitions will be familiar to management applications and the people using them. For instance, use the DMTF-approved System MIF to describe a desktop system or server rather than creating a new, proprietary MIF.

Your product model must allow for different views of the same data by providing both standard (public) groups of attributes and private (individual) groups of attributes. For instance, both an external fax modem and an internal data modem can be described by one or more groups describing characteristics shared by all modems. But each individualized product MIF would also need some groups of attributes that describe characteristics that apply only to each specific variant: internal versus external, fax modem versus data-only modem.

Basics of good MIF modeling

A good MIF file contains groups that are standard, consistent, modular, scalable, flexible and reusable. To design a well-formed MIF definition, follow these principles:

- **Make a detailed list of the manageable attributes for your product.** At first, don't worry about classifying the attributes. *After creating this initial list, look at MIF files for similar products or in related Standard Groups MIF files for groups and attributes that suit your product well.*

- **Use the DMTF-approved Standard Groups MIF definitions whenever you can, rather than creating a private, non-standard group.** When you use a Standard Group, change only the attribute values for all attributes in that group. If you don't want to use one or more of its attributes, use the ***unsupported*** keyword in that attribute's Value statement. *Do not change any other items in the standard attributes, such as AttributeID or type. Do not renumber the attributes within a Standard Group.*

- **Do not add your own private attributes to Standard Groups.** Instead, use the Standard Group and add a separate, private group of your own definition to add attributes to the MIF. *If you change the Standard Group by adding private attributes, it is no longer a Standard Group.*

- **Make private groups reusable, if possible.** Since the private groups you define will be proprietary to your product, try to develop groups that can be standardized for use with other products you may develop. Keep all of your similar products in mind, even if you are only writing a MIF for *one* of the products. *Standardizing your private group definitions reaps the same manageability benefits for your products*

that using the DMTF-approved Standard Groups definitions bring to general interoperability of DMI systems.

- **Make private groups small.** The accepted practice is to use 20 or fewer attributes per group. As long as the group makes sense, it is even reasonable for it to have only one or two attributes. *Several small groups are easier to manage, more modular and easier to reuse than a few large groups.*

- **Use consistent naming and terminology** for groups and attributes so they are easier to reuse.

- **Organize your MIF files consistently.** *If you produce more than one kind of component,* this allows you to reuse proprietary MIF groups from one product to another, saving time and effort.

- **Use enumerated lists liberally.** Enumeration can improve the readability of your MIF data in DMI browsers, simplify efforts to translate your MIF data into other languages, and enhance your key lists and attribute values for data in keyed groups (tables).

- **Use meaningful keylists.** Index tables with attributes that make sense for the component rather than an arbitrary indexing value. Use more than one key, if necessary. *Additional keys allow multiple views of the same data and provide more flexibility for management applications.*

- **Pack as much meaningful information into your descriptions as possible.** Description statements can be a very powerful help tool; *use as many lines as needed.*

- **Include minimum, maximum and default** values in Description statements when possible.

- **Try to identify the possible audiences for your MIFs and address their needs in your descriptions.** Remember that different audiences will be reading the descriptions; *you want to make your product as easy to manage as possible.*

Questions to answer during the modeling phase

When modeling your component, the answers to these questions will help develop effective MIF data:

- Is there an existing standardized MIF file or group definition that I can use when describing all or part of this component?
  Tip: *Use existing MIF files and Standard Groups MIF definitions as much as possible, to allow more management applications to intelligently manage your component.*

  ⇒ Can I use existing DMTF-approved Standard Groups definitions, such as the DMTF definitions for “System Resources” and “Operational Status” groups?
  Tip: *Use as many of the Standard Groups definitions as are applicable, to allow more management applications to intelligently manage your component.*

⇒ Are there existing private (proprietary) MIF files or groups I can draw from?
Tip: *Existing MIF files may already have addressed difficult aspects of modeling your own product.*

- What are the vendor-specific aspects of this component to be modeled?
- What is a sensible way to group the manageable aspects of this product, especially the vendor-specific or proprietary attributes?
- What information needs to be included to allow easy asset management?
Tip: *Be sure to include such things as product version number or release number, and serial number.*

  Which attributes should not change? *These attributes have read-only accessibility.* Which should be changeable? Of those that are changeable, are there any where the value should not be seen by the management application, although it can be changed? *This determines whether attributes are accessible as read-write or write-only.*
- Does this attribute require a single value or does it require a table? That is, is it necessary to provide for multiple instances of that attribute's value to correctly model the component?
Tip: *Generally, tables of values are more flexible and allow easier expansion later.*
- If this attribute is part of a table of values, what is the most sensible way to key into that table? Is more than one key needed?
Tip: *More than one key allows for multiple views of the same data, and provides more flexible manageability.*
- Will the value for this attribute be provided by the component through the direct interface, or kept in the database?
Tip: *Remember that when values are kept only in the database, it can be difficult to keep them accurate and reliable because they rely on management applications for their values (which operate independently of each other). Instrumentation allows more up-to-date and reliable data.*

MIF file writing tips

- Model your product as described above.
- In defining your groups, determine which will be DMTF-approved Standard Groups definitions and which will be private (proprietary) groups. Use as many existing Standard Groups as possible. *The first group is always the Component ID group.*
- Determine which groups will be tabular and how the tables will be keyed.
- Define any enumeration that may be needed for the component as a whole.
- Determine the attributes to be defined in each private group.
- For each attribute:

  ⇒ Choose a name (remember to use consistent naming techniques).

  ⇒ Assign it an ID number, unique and sequential within its group.

⇒ Write a meaningful description.

⇒ Determine what type of data will be used (integer, gauge, counter, display string, octet string or date).

⇒ Determine whether the attribute's access method should be read-only, read-write, or write-only.

⇒ Determine whether the attribute's storage location should be common or specific.

⇒ Decide how the attribute's value will be derived (from within the database, by accessing the direct interface or from a runtime program).

⇒ Decide whether the attribute's value is a single value or is part of an indexed table of values.

⇒ If it is part of an indexed table, decide whether this attribute should contribute to the indexing key(s).

- Follow the MIF grammar and syntax as described in the *DMI Specification*. Be sure the file is a simple ASCII text file with no embedded formatting commands or invisible characters.

It doesn't matter in what order the groups appear in the MIF file. The information should be organized in a way that makes sense so it can be checked, expanded or changed over time.

Checking the MIF file

Once you've written the MIF file, you must check that its grammar parses correctly so it can be installed in the MIF database. You might also want to check that the MIF data conforms to standardized groups definitions, such as the DMTF-approved Standard Groups definitions or your own in-house private group definitions. This SDK provides the DMI 2.0 MIF Conformance Checker (COMPCHK2) tool to check MIF syntax and benchmark your MIF files. *(See Chapter 8 of this Reference manual.)*

Remember: If Standard Groups definitions are used, all attributes included in that group must be used, even if some of the attributes are not supported. Never delete attributes from or add attributes to a Standard Group. Use private groups for vendor-specific information.

Other MIF developer resources

Each of the DMI 2.0 code samples has an instrumented component MIF file for you to examine. There is also a Sample MIF in Section 2.3 of the *DMI Specification*. The COMPCHK2 tool includes a template file, MASTER.MIF, that contains DMTF-approved Standard Groups definitions. These MIF examples demonstrate the following MIF definitions:

- path statements
- enumeration
- tabular data
- read-only and read-write attributes
- instrumented attributes
- unsupported attributes
- private groups

Note: The Standard Groups definitions may change over time; always check with the DMTF home page on the World Wide Web to get the latest definitions of their Standard Groups. *(See Chapter 1 of this Reference Manual for contact information.)*

Chapter 5 - DMI 2.0 Code Samples

Overview

This chapter describes some code samples that use the procedural MI and CI interfaces of a DMI 2.0 Service Provider. Each sample includes examples of instrumented MIF files, direct interface instrumentation code and management applications.

These samples also present methods of ensuring that the component and its instrumentation are correctly installed and running. The first set of samples, Trusted Code, ensures that an instrumented component is operational by enlisting the management application to determine whether its component MIF is present (and installing it when needed), and confirming that its component instrumentation is resident and registered with the DMI-SP. The second set of samples, Multi-Timer, relies on the user to load the instrumentation application, which is enlisted to ensure that its component MIF has been installed and registered; once the instrumentation application confirms that it has succeeded, the user runs the Multi-Timer management application.

For each of the code samples, this SDK has provided ready-to-use executables (look for their icons in the DMI SDK program group) and source code. (All the sample source code included with this SDK were developed using Microsoft Visual C++ version 4.2.) Each sample provides a framework for creating DMI direct interface instrumentation and demonstrates how it interfaces to the DMI Service Provider. Each code sample section provides event diagrams of its operation and troubleshooting tips. Study the description of the source code, how to build each executable sample and how each sample should operate.

Trusted Code samples

This sample demonstrates how component instrumentation can be implemented using Windows dynamic-link libraries (DLL). In this example, the component instrumentation code is provided through a DLL, which is loaded by the management application when it starts and unloaded when the management application terminates.

The executables included in this sample

- Trusted Management Application (TRUSTDI.EXE),
- Trusted Component MIF (TRUSTDI.MIF), and
- Trusted Direct Interface Instrumentation API (TRUSTDDI.DLL).

TRUSTDI.EXE is a Windows-based component installation program that installs the TRUSTDI.MIF, loads the TRUSTDDI.DLL library module and registers the instrumentation with the DMI-SP. *Note that the TRUSTDI program does not send management requests.* The instrumentation module, TRUSTDDI.DLL, is implemented as a Windows dynamic-link library.

The Trusted Component's MIF data

The Names of groups within the TRUSTDI.MIF describe either the kind of group it is or the Access type of its Values. When a particular DMI data keyword appears in the name, it describes either the types of attributes used in its key list or the use of a special keyword in its Value statements.

TRUSTDI.MIF defines these groups:

- Scalar Data Types
- Mixed Access Types
- Unsupported Values
- Unknown Values
- Counter32 Keyed Table
- Counter64 Keyed Table
- Gauge Keyed Table
- Integer32 Keyed Table
- Integer64 Keyed Table
- Octet Keyed Table
- Display String Keyed Table
- Date Keyed Table

Building the Trusted Code samples

The source files and executables for the Trusted Code samples are in the subdirectory *%WIN32DMIDIR%*\EXAMPLES\TRUSTDI.

The source files included in this sample

- **TRUSTDI.C,** the main source file for the TRUSTDI management application and its user interface, responsible for loading the instrumentation (DLL) into memory and then registering each of the DLL's five PUBLIC functions with the DMI Service Provider. When compiled, it produces the TRUSTDI.EXE management application.
- **TRUSTDI.H,** the header file for TRUSTDI.C module.
- **COMPOSER.ICO,** the icon file for the TRUSTDI.EXE management application.
- **TRUSTDI.MIF,** the Trusted Component MIF file, which must be installed into the component MIF database to activate the instrumentation.
- **TRUSTDI.MDP,** the Visual C++ 4.2 workspace file used to build the management application.
- **TRUSTDDI.C,** the dynamic-link library's source file, which implements the direct interface instrumentation. When compiled, it produces the TRUSTDDI.DLL program.
- **TRUSTDDI.H,** the header file for the TRUSTDDI.C module.
- **RESOURCE.H,** a header file that defines the user interface constant identifiers.
- **TRUSTDDI.MDP,** the Visual C++ 4.2 workspace file used to build the DLL module.

Building Trusted Code samples with Visual C++* v4.2 IDE

To build the Trusted Component modules, perform these steps:

1. Run the Microsoft Visual C++ 4.2 Development Studio program.
2. Load the TRUSTDDI.MDP workspace (source code for the TRUSTDDI.DLL module).
3. From the BUILD menu, select SETTINGS. Choose the Preprocessor option from the C/C++ Tabbed dialog box. Set the Additional Include Directory entry to the INCLUDE directory of the DMI Service Provider installed on your system.

4. Replace the WCDMI.LIB module in the project files list with the one in the LIB directory of the DMI Service Provider installed on your system.
5. From the BUILD menu, select UPDATE ALL DEPENDENCIES.
6. Rebuild the project to generate the TRUSTDDI.DLL module.
7. Repeat Steps 2 through 6 to build the management application (with appropriate filename changes: load TRUSTDI.MDP to generate the TRUSTDI.EXE installation program).
8. **Verify that the TRUSTDDI.DLL module and the TRUSTDI.MIF file are in the same directory as the TRUSTDI.EXE module.**
9. Run the TRUSTDI management application.
10. Open the TRUSTDDI.C module and set breakpoints on the five instrumentation entry points: *GetAttribute(), SetAttribute(), GetNextAttribute(), AddRow(), DeleteRow().*
11. Run the DCTS2 test tool to view the attribute values in different groups and tables in the "Trusted Code Sample" component MIF data.

When you try to view the value of an attribute within a group using a tool like DCTS2 to browse attribute values, the Service Provider will call the entry points of the TRUSTDDI.DLL instrumentation and break on the breakpoints that you set in Step 10 above.

Using the Trusted Code samples

In this sample, the management application has been designed to ensure that the Trusted Component MIF (TRUSTDI.MIF) is installed and the Trusted Direct Interface (DI) Instrumentation (TRUSTDDI.DLL) is resident. The Trusted DI Instrumentation sample demonstrates how a component manufacturer can provide a direct interface for passing information between the instrumentation and the Service Provider.

When the TRUSTDI application loads, it loads the TRUSTDDI.DLL module into memory. Once the TRUSTDDI.DLL module has been loaded, the TRUSTDI application tries to find the Trusted Component MIF in the local component MIF database. If the TRUSTDI.MIF is not yet installed in the MIF data store, the TRUSTDI application will install it into the DMI-SP MIF database. Once the TRUSTDI.MIF is installed, the TRUSTDI application registers the Trusted DI instrumentation entry points with the Service Provider.

The TRUSTDDI.DLL, the TRUSTDI.MIF and the TRUSTDI.EXE files must all be in the same directory to use this sample; the SDK Setup program puts them in the *%WIN32DMIPATH%*\EXAMPLES\TRUSTDI directory.

To use this sample, start the Trusted Management Application by either double-clicking the Trusted Management Application icon or choosing Run from the START menu and browsing for:

TRUSTDI.EXE

The Trusted Management Application window has no interactive management options to exercise the Trusted Component. To do this, you can use the DCTS2 tool to browse the Service Provider's MIF database and interact with the Trusted Component.

To start DCTS2, double-click the DMI Component Test System icon or choose Run from the START menu, then browse the .\DCTS subdirectory[11] for:

DCTS2.EXE

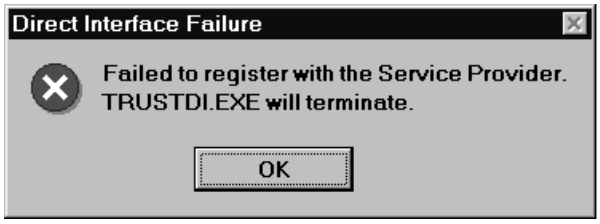
Define a machine connection for your local system and then select the Service Provider database option from the DATABASE menu to open a Service Provider database Browser window. Connect the Service Provider database browser to your local system. Look for the component named Trusted Component Example. If it is in the database, then your MIF file was correctly installed. To quit DCTS2, select Exit from the FILE menu.

To quit the TRUSTDI application, select Exit from the FILE menu. Closing the TRUSTDI application will automatically unregister and unload the TRUSTDDI instrumentation.

The Trusted Management Application

Because this sample demonstrates a direct interface between instrumentation and the Service Provider, the TRUSTDI.EXE application first loads the TRUSTDDI.DLL API and then dynamically determines the ComponentID for the TRUSTDI component. If the component is installed in the component MIF database, the program retrieves its ComponentID. Once the ComponentID is known, the TRUSTDI application registers the component with the Service Provider as direct interface. If the component is not installed, the program installs it using the TRUSTDI.MIF file and then registers the component with the Service Provider.

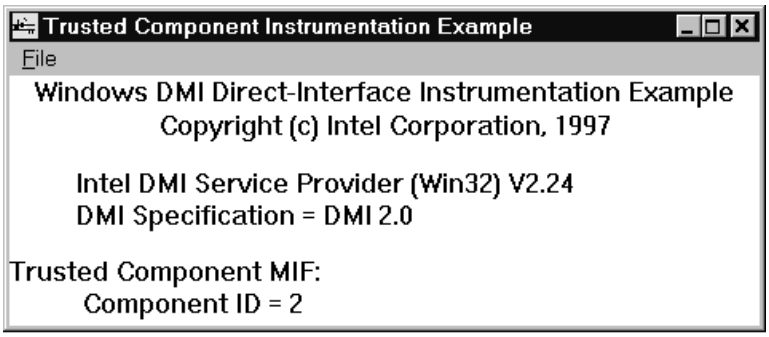
TRUSTDI reports the results of instrumentation registration in one of the following ways:



If the component registration fails, this message box appears. If you click OK, the program terminates.

[11] The .\DCTS subdirectory is a peer to the *%WIN32DMIPATH%* subdirectory.

If the component is registered successfully with the Service Provider, the program displays its main window. The Value of the ComponentID is dynamically derived when the MIF is installed.



Note: The values that display for the Version and the ComponentID may be different from those shown, depending of the version of the DMI Service Provider in use and the configuration of the DMI-SP database.

Exercising the Trusted DI instrumentation

After the component is successfully registered with the local DMI Service Provider, you can make queries to the DMI-SP to get attribute values in different database groups. When the DMI-SP receives a request, it will call one of the TRUSTDDI.DLL entry points—*GetAttribute(), SetAttribute(), GetNextAttribute(), AddRow(), DeleteRow()*—to service the request. Use DCTS2 to connect to the local system, and open a Browser window into the Service Provider database to access the instrumented attributes of the Trusted Component.

How to exercise the Trusted Component

To use the direct interface example, perform the following steps:

1. Verify that the WIN32SL.EXE service is loaded and is running.
2. Run the TRUSTDI.EXE program.
3. Run the DCTS2 program. Create a connection to your local system. Open a Browser window onto the DMI Service Provider database on your local system. Connect the Service Provider database browser to your local system. Expand the Trusted Component Example component by double-clicking the tree node box labeled C, located to the left of the component's name.
4. Expand the different groups that are displayed on the left pane of the Browser window to view the attribute values in each group. Click an attribute to access its Value (displayed on the right pane of the Browser window) and exercise the instrumentation. Expected results for each group's attributes are:
   - The "Mixed Access Types" group; what the DMI-SP returns depends on what the given attribute's Access property is set to:
     ⇒ *Write-Only*—"Attribute is write only" error is returned to caller
     ⇒ *Read-Only*—Successful access (but error if you attempt to Set the Value)
     ⇒ *Read-Write*—Successful access

- The “Unsupported Values For Each DMI Type” group: “Attribute Not Supported” is returned by the DMI-SP; DCTS2 displays “Unsupported” in its Browser’s Access/Storage field; other DMI browsers may display “Attribute Not Supported.”
- The “Unknown Values For Each DMI Type” group: “Attribute Unknown” is returned by the DMI-SP; the attributes in this group have their Value property set to “Unknown.”
- All other groups return type-appropriate Values successfully.

Exiting the Trusted Code samples

When you exit the TRUSTDI application, the program queries the TRUSTDDI.DLL to see if the instrumentation is still processing a request. When the DLL is no longer busy, the TRUSTDI application unregisters the component, unloads the TRUSTDDI.DLL from memory and then exits. To remove the Trusted Component from the DMI MIF database, use DCTS2 to uninstall the Trusted Component MIF data.

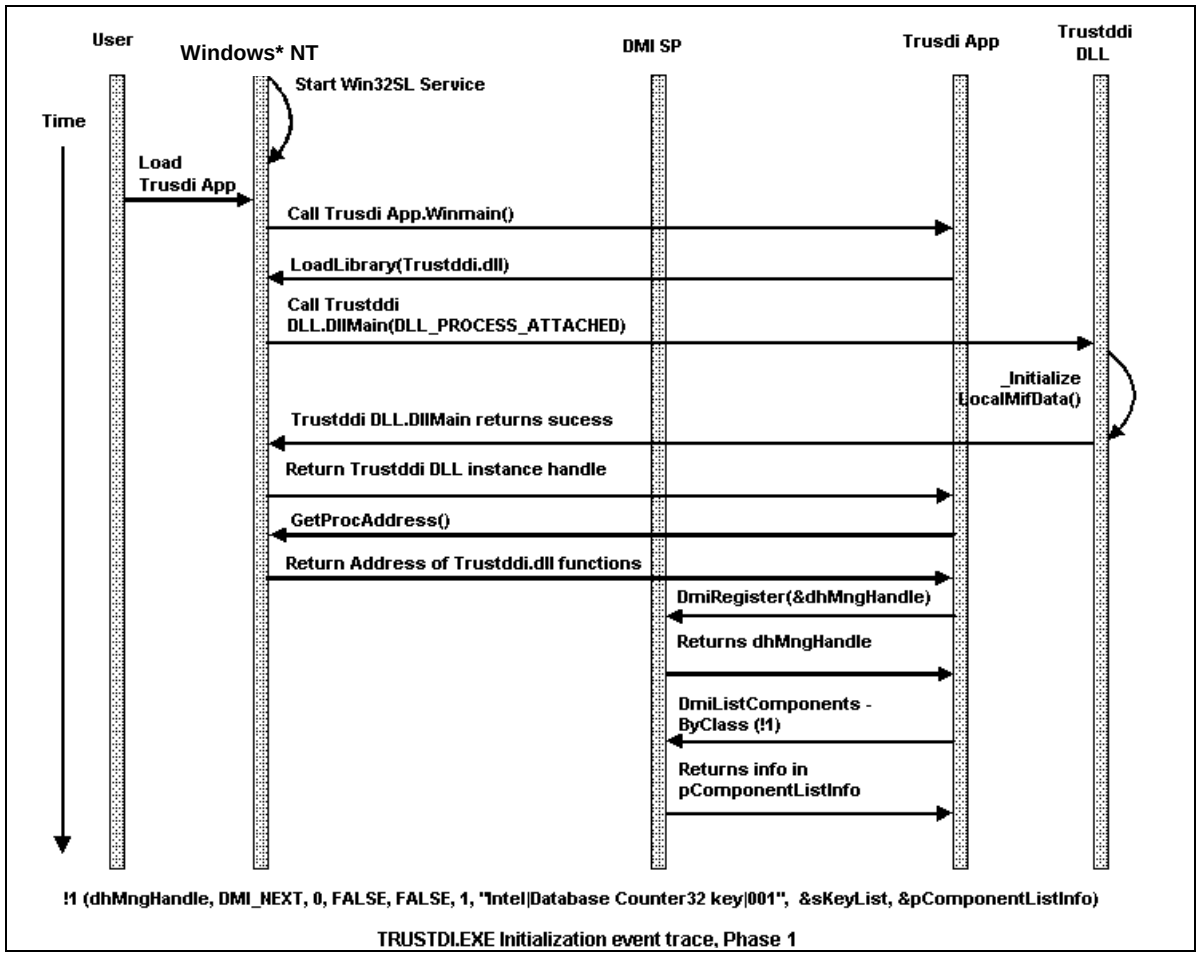
Trusted Code samples event diagrams

The following diagrams show the events that take place at various stages of the program initialization, operation and termination.

Initialization of the TRUSTDDI application, Phase 1

The following sequence of operations takes place during Phase 1 of the initialization of TRUSTDI.EXE:

1. During the system initialization, the operating system loads, then starts the WIN32SL.EXE Service.
2. The user invokes the TRUSTDI installation application.
3. The operating system loads and runs the TRUSTDI installation application.
4. The TRUSTDI installation application makes a request to the operating system to load the TRUSTDDI.DLL library module by calling the *LoadLibrary()* function.
5. The operating system calls the TRUSTDDI.DLL module’s *DllMain()* function with a DLL_PROCESS_ATTACHED message.
6. Upon receiving the DLL_PROCESS_ATTACHED message, the TRUSTDDI.DLL module calls its private *_InitializeLocalMifData()* method to initialize its internal data structures.
7. The TRUSTDDI.DLL module returns a status of success to the operating system.
8. The operating system returns the DLL’s instance handle to the TRUSTDI application.
9. Using the DLL’s instance handle, the TRUSTDI application passes a request to the operating system for the addresses of the TRUSTDDI.DLL module’s instrumentation entry points by making several calls to the *GetProcAddress()* function.



TRUSTDI.EXE Initialization event trace, Phase 1

10. The operating system returns the addresses of the DLL's entry points to the TRUSTDI application.

11. The TRUSTDI application registers with the DMI-SP as a management application by calling the DMI-SP's *DmiRegister()* function.

12. The DMI-SP returns a management handle to the TRUSTDI application.

13. Using this management handle, the TRUSTDI application requests the DMI-SP to see if the TRUSTDI component has been installed by passing the class string of the Counter32 Keyed Table Group as a filter.

14. The DMI-SP returns the query information to the TRUSTDI application.

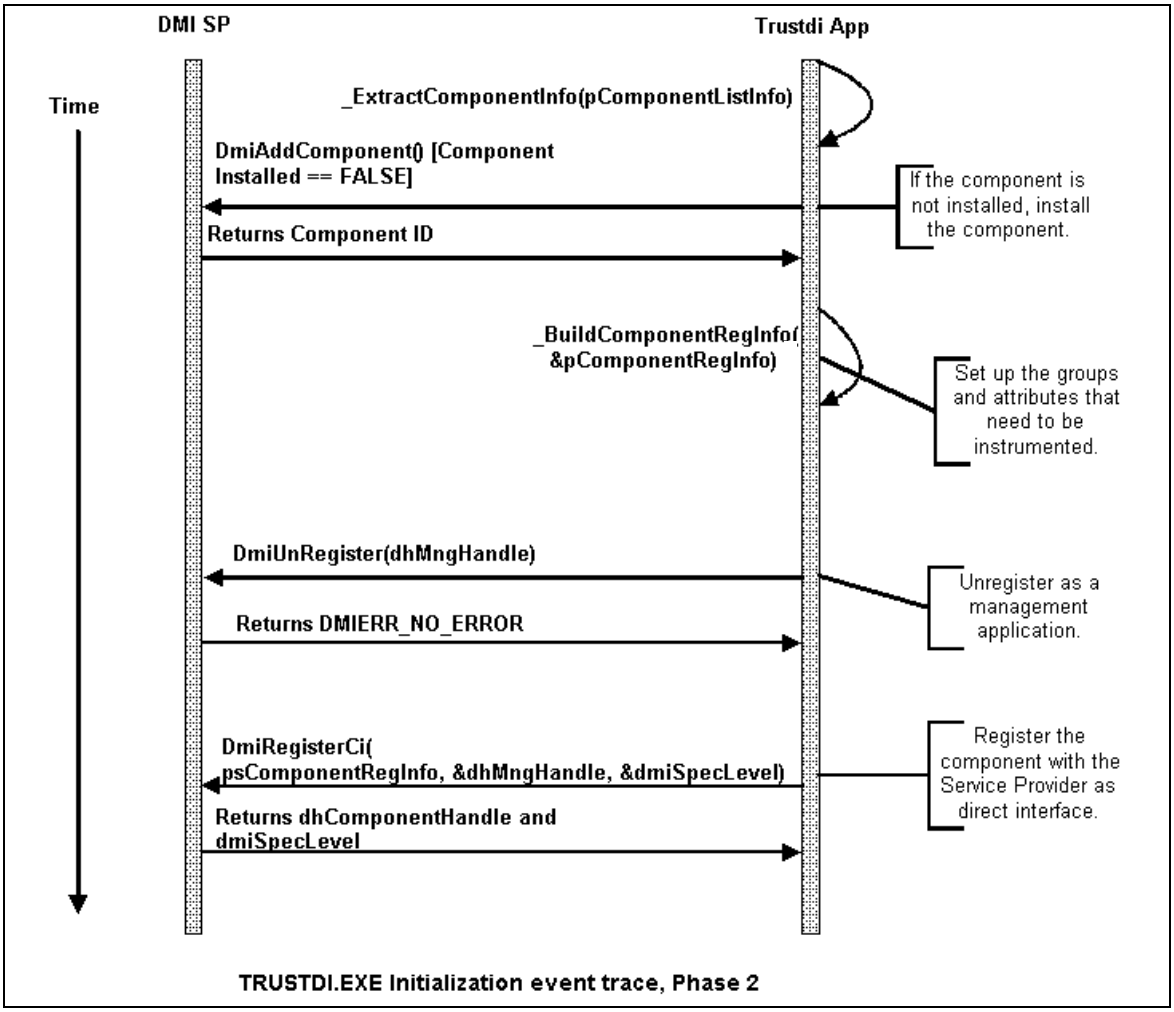
Initialization of the TRUSTDI application, Phase 2

The following sequence of operations takes place in Phase 2 of the initialization of TRUSTDI.EXE:

1. The TRUSTDI application calls its private function, *_ExtractComponentInfo()*, passing the component list information received previously from the DMI-SP. The *_ExtractComponentInfo()* function extracts component information from the

component list provided by the DMI-SP to determine whether the TRUSTDI component has been installed. If the component has been installed, then the TRUSTDI application captures the ComponentID value.

2. If the component is not yet installed, the TRUSTDI application sends a component installation request to the DMI-SP by calling the *DmiAddComponent()* function.
3. The DMI-SP installs the component into the database and returns the component ID to the TRUSTDI application.
4. The TRUSTDI application calls its private function, *_BuildComponentRegInfo()*, to set up the groups and attributes in the TRUSTDI.MIF file that need to be instrumented.



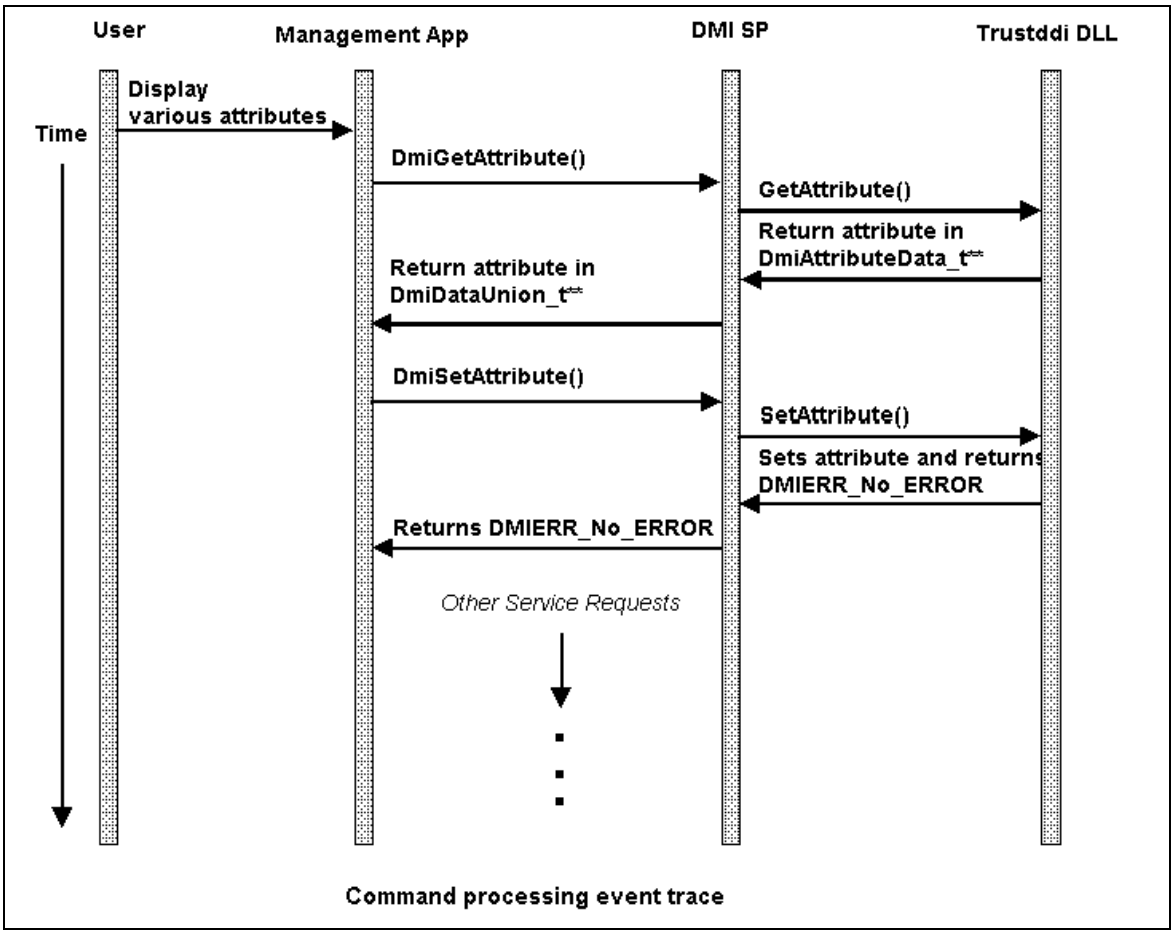
5. Using the management application handle, the TRUSTDI application sends an unregister request the DMI-SP by calling the DMI-SP's *DmiUnregister()* function.
6. DMI-SP unregisters the management application handle assigned to TRUSTDI.EXE and returns a status of 'success.'
7. The TRUSTDI application makes a request to the DMI-SP to register the component instrumentation as direct interface by calling DMI-SP's *DmiRegisterCi()* function.

8. DMI-SP registers the component instrumentation as direct interface, and returns the component handle and the *DMI Specification* version of the DMI-SP to the TRUSTDI application.

Command processing with the TRUSTDDI instrumentation

The following sequence of operations takes place during command processing:

1. Using a management application such as DCTS2, browse the database to view the Value and Description of various attributes.
2. For example, to view an attribute's Value, the management application calls the DMI-SP's *DmiGetAttribute()* function, passing the GroupID, AttributeID and a key list (optional) for the attribute of interest.
3. Since the component is instrumented, the DMI-SP calls the TRUSTDDI.DLL entry point *GetAttribute()* function to get the attribute's data.

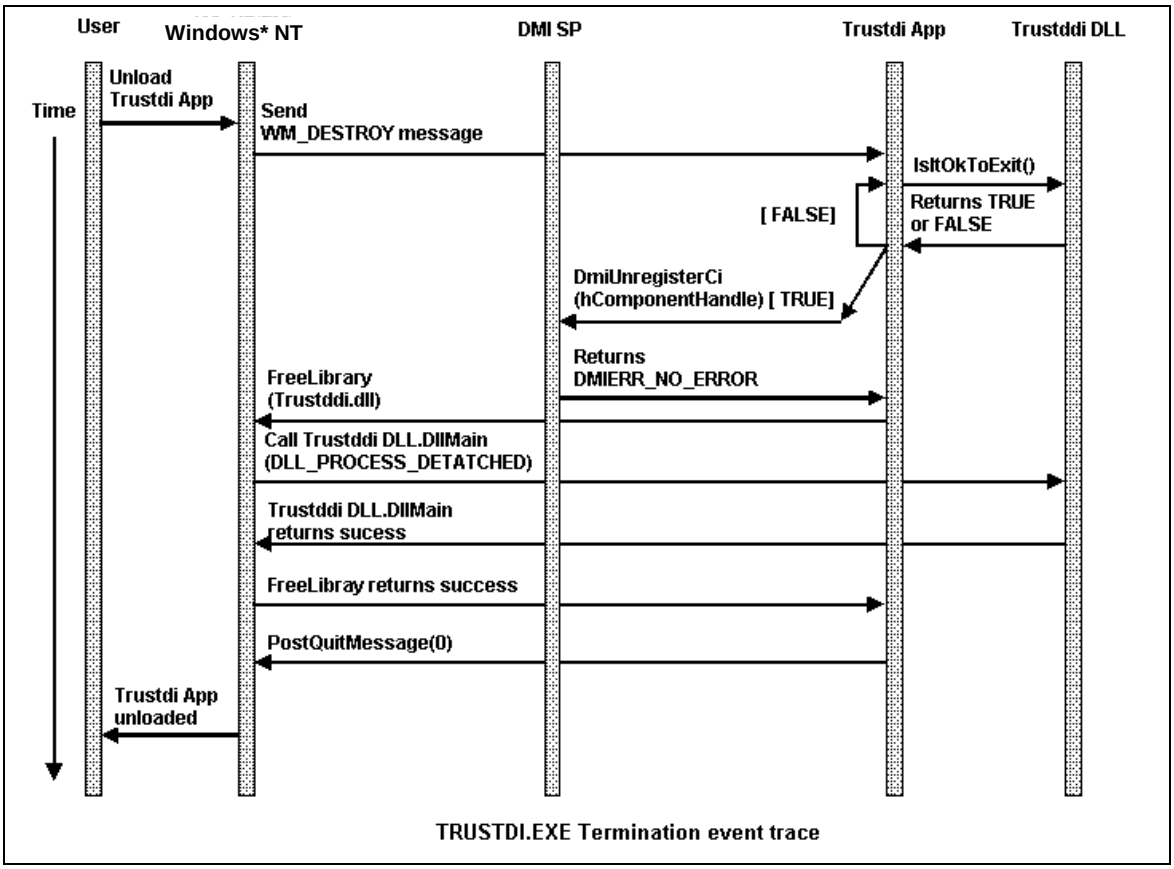


4. The TRUSTDDI.DLL module returns the attribute's Value to the DMI-SP.
5. Similar operations take place for setting the Value of an attribute.

Terminating the TRUSTDI application

The following sequence of operations takes place during the termination process:

1. The user chooses the Exit option of the TRUSTDI application.
2. The operating system sends the WM_DESTROY message to the application.
3. Before terminating itself, the TRUSTDI application queries the TRUSTDDI.DLL module's status by calling its public function, *IsItOkToExit()*.
4. If the DLL module is idle, it returns TRUE, indicating that it's OK to terminate and unload the DLL from memory. If the DLL is still processing a Service request, it returns FALSE.
5. If the TRUSTDI application receives TRUE from the DLL, it submits a request to the DMI-SP to unregister the component instrumentation as direct interface by using the component instrumentation handle when calling the *DmiUnregisterCi()* function.



6. DMI-SP unregisters the component and returns a status of 'success' to the TRUSTDI application.

7. Using the DLL's instance handle, the TRUSTDI application makes a request to the operating system to unload the TRUSTDDI.DLL module from memory by calling the *FreeLibrary()* function.

8. Using a DLL_PROCESS_DETACH message, the operating system calls the *DllMain()* function of TRUSTDDI.DLL, which allows the DLL to do cleanup before it unloads from memory.

9. The TRUSTDDI.DLL module returns a status of 'success' to the operating system.

10. The operating system returns a status of 'success' to the TRUSTDI application, indicating that the DLL has been successfully unloaded.

11. The TRUSTDI application sends a *PostQuitMessage()* to the operating system, indicating that it is ready to terminate.

12. The operating system unloads the TRUSTDI application from memory.

Troubleshooting runtime problems

To debug the Trusted Management Application, you can execute the TRUSTDI.EXE program under the Microsoft Visual C++ 4.2 development environment and set breakpoints on the five instrumentation functions defined in the TRUSTDDI.C module. Use DCTS2 to access the instrumented attributes.

Stepping through the Trusted Code sample with a debugger

To debug the TRUSTDI application and the TRUSTDDI instrumentation, perform the following steps:

1. Run the Microsoft Visual C++ 4.2 Development Studio program.
2. Load the TRUSTDDI.MDP workspace.
3. Run the TRUSTDI application.
4. Open the TRUSTDDI.C module and set breakpoints on the five instrumentation entry points: *GetAttribute(), SetAttribute(), GetNextAttribute(), AddRow(), DeleteRow().*
5. Click **OK** when the program displays the message box indicating that the component has been successfully registered.
6. After invoking DCTS2, connect to the local system and open a Browser into the local system's Service Provider database. Expand the component tree, then click to view the attribute values in different groups and tables in the TRUSTDI.MIF component MIF file.

Note: **You'll need to copy the TRUSTDI.MIF file into the same directory that the Development Studio's debugger copied the TRUSTDI.EXE file.** When you use a browsing tool like DCTS2 to access attribute values, the Service Provider will call the entry points of the TRUSTDDI.DLL instrumentation, and the debugger "break" on the breakpoints that you set in Step 4 above.

Multi-Timer samples

As another example of direct interface instrumentation that uses the procedural Component Interface, we have included a program file which manages a table of Timers. The Multi-Timer samples are designed around a component whose instrumentation can use the time functions of the system running the Multi-Timer Instrumentation to trigger indications to the system's Service Provider.

The model of the Multi-Timer example is based on the idea of a count-down interval timer, like that on a kitchen stove or range. In this case, the triggering events are the passage of time (seconds) and the moment the Timer counts down to 0 (zero). Timers may be added, deleted and edited, and count at the rate of approximately 1 tick per second. This example uses indications (to notify the Multi-Timer Management Application, for example) when a Timer has counted down to zero or has been deleted from the timer list.

The files included in this sample

- Multi-Timer Direct Interface Instrumentation Application (MTIMERDI.EXE)
- Multi-Timer Component MIF (MTIMERDI.MIF)
- Multi-Timer Management Application (MTIMERMA.EXE)

MTIMERDI.EXE is a Windows-based instrumentation installation program that installs the MTIMERDI.MIF and registers itself as direct interface instrumentation with the DMI-SP. MTIMERMA.EXE is a Windows-based management application that allows the user to interact with the Multi-Timer Component, to add and manipulate Timers.

The Multi-Timer's MIF data

The Names of groups within the MTIMERDI.MIF define the straightforward purpose of their attributes. The groups defined in MTIMERDI.MIF are:

- Component ID
- Multi-Timer Table
- Multi-Timer Control
- Event Generation

Building the Multi-Timer samples

The source files and executables for the Trusted Code samples are in the subdirectory *%WIN32DMIDIR%*\EXAMPLES\MTIMER.

The source files included in this sample

- **MTIMERMA.C,** the main source file for the MTIMERMA management application and its user interface. When compiled, it produces the MTIMERMA.EXE management application.

- **MTIMERMA.H,** the header file for the MTIMERMA.C module that defines the GroupIDs and AttributeIDs for the MTIMERDI.MIF and a variety of other symbols used in the application.

- **MDMIFUNC.C,** the source file that contains a collection of PUBLIC and PRIVATE functions to support DMI operations. This file is largely operating system-independent.
- **MDMIFUNC.H,** the header file for the MDMIFUNC.C module; it exports a variety of functions and variables used in the management application for performing DMI operations. (This header is automatically <included> by MTIMERMA.H.)
- **RESOURCE.H,** a header file that defines constant identifiers for the management application's user interface.
- **MTIMERMA.ICO,** the icon file for the MTIMERMA.EXE management application.
- **MTIMERMA.MDP,** the make file for this application. This file assumes that the MSDEV tool kit is installed on drive C.
- **MTIMER.RC,** the resource file for this management application.
- **MTIMERDI.MIF,** the Multi-Timer Component MIF file, which must be installed into the component MIF database to activate the instrumentation.
- **MTIMERDI.C,** the main source file for the Direct Interface instrumentation for the Multi-Timer example and the user interface for its installation application. When compiled, it produces the MTIMERDI.EXE program.
- **MTIMERDI.H,** a header file for the MTIMERDI.C module; it defines a variety of constants and data types used by the instrumentation.
- **DIRESOUR.H,** a header file that defines constant identifiers for the instrumentation's user interface.
- **MTIMERDI.ICO,** the icon file for the MTIMERDI.EXE instrumentation.
- **MTIMERDI.RC,** the resource file for the instrumentation.
- **MTIMERDI.MDP,** the make file for the instrumentation. This file assumes that the MSDEV tool kit is installed on drive C.

Steps to build Multi-Timer samples with Visual C++* v4.2 IDE

To build the Multi-Timer modules, perform the following steps:

1. Run the Microsoft Visual C++ 4.2 Development Studio program.
2. Load the MTIMERDI.MDP workspace.
3. From the BUILD menu, select SETTINGS. Choose the Preprocessor option from the C/C++ Tabbed dialog box. Set the Additional Include Directory entry to the INCLUDE directory of the DMI Service Provider installed on your system and to the directory containing the header files for the Multi-Timer samples on your system.
4. Replace the WCDMI.LIB module in the project files list with the one in the LIB directory of the DMI Service Provider installed on your system.

5. From the BUILD menu, select UPDATE ALL DEPENDENCIES.
6. Rebuild the project to generate the MTIMERDI.EXE module.
7. Repeat Steps 2 through 6 to build the management application (with appropriate filename changes: load MTIMERMA.MDP to generate MTIMERMA.EXE).
8. Verify that the MTIMERDI.MIF data file is in the same directory as the MTIMERDI.EXE module.
9. Run the MTIMERDI.EXE instrumentation installation application. (The window may be minimized, but this application must remain resident for the instrumentation to operate.)
10. Open the MTIMERDI.C module and set breakpoints on the five instrumentation entry points: *GetAttribute(), SetAttribute(), GetNextAttribute(), AddRow(), DeleteRow().*
11. Run the DCTS2 tool and open its Event Monitor window to capture information about events as they happen.
12. Run the MTIMERMA.EXE management application. (You can use either this application to exercise the instrumentation by setting timers, or exercise the instrumented attributes in the DCTS2 tool by opening a Browser window for the Service Provider database on the local system connection and clicking the "Windows Multi-Timer Example" component's attributes.)
13. Set up one or more Timers, which will cause indications as they count down to 0 (zero) or are deleted.

When trying to view the value of an attribute within a group using a tool like DCTS2 to browse attribute values, the Service Provider will call the entry points of the instrumentation MTIMERDI.EXE and break on the breakpoints that you set in Step 10 above.

Using the Multi-Timer samples

The Multi-Timer Direct Interface Instrumentation (MTIMERDI.EXE) demonstrates how a component manufacturer can provide direct interface instrumentation that sends asynchronous indications to the Service Provider for 32-bit Windows systems. When the Multi-Timer's instrumentation code (MTIMERDI.EXE) loads, it installs the Multi-Timer Component MIF (MTIMERDI.MIF) if it can't find an instance of the Multi-Timer component in the active component MIF database.

To use this example, first verify that both the MTIMERDI instrumentation and the Multi-Timer Component MIF file are in the same directory. Before running the management application, you must first run the Multi-Timer direct interface instrumentation. Double-click the Multi-Timer Instrumentation icon, or choose Run from the START menu and browse the ***%WIN32DMIPATH%*\EXAMPLES\MTIMER** directory for:

MTIMERDI.EXE

If the MTIMERDI.MIF is not yet installed in the MIF data store, the MTIMERDI instrumentation will install it into the DMI-SP. Minimize the MTIMERDI instrumentation

on the desktop, if desired. Browse the component list with the DCTS2 tool to make sure the MIF file was correctly installed.

To start DCTS2, double-click the DMI Component Test System icon, or choose Run from the START menu, then browse the .\DCTS subdirectory[12] for:

DCTS2.EXE

Define a machine connection for your local system and then select the Service Provider database option of the DATABASE menu to open a Browser window. Connect the Service Provider database browser to your local system. Look for the component named Windows Multi-Timer Example. If it is in the database, then your MIF file was correctly installed. To quit DCTS2, select Exit from the FILE menu.

Once the direct interface instrumentation is loaded, it can be minimized on the desktop while you are running the Multi-Timer program. Run the management application and try out the example. Double-click the Multi-Timer Sample icon, or choose Run from the START menu, then browse the directory *%WIN32DMIPATH%*\EXAMPLES\MTIMER for:

MTIMERMA.EXE

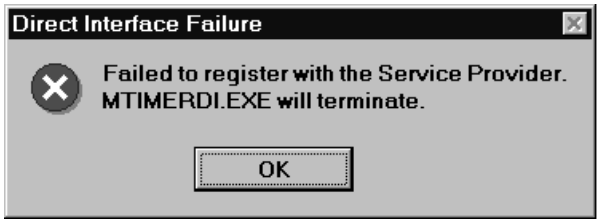
The Multi-Timer Management Application window opens with three menus: FILE, TIMER and ABOUT. From the TIMER menu, select Add; a dialog window prompts you for a Timer number and an initial time interval (in seconds). The MTIMERDI instrumentation sends an indication whenever a Timer's interval has elapsed to 0 or a Timer has been deleted. When the MTIMERMA management application receives an indication from the Service Provider, it displays a message window to alert the user that the Timer has issued its indication.

From the FILE menu, choose Exit to quit the MTIMERMA application. To shut down the MTIMERDI instrumentation, maximize its window and choose Exit from the FILE menu to remove it from the system.

The Multi-Timer Management Application

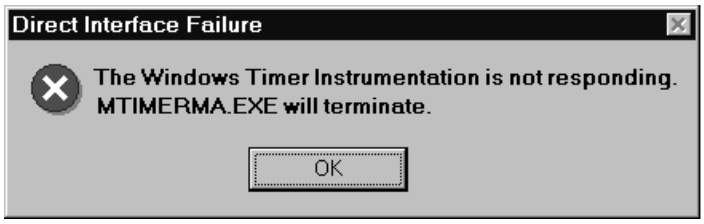
The Multi-Timer Management Application does *not* perform any automatic installation of the Multi-Timer Component MIF, nor its instrumentation. Therefore, you must invoke the MTIMERDI.EXE instrumentation before you invoke the MTIMERMA.EXE management application.

[12] The .\DCTS subdirectory is a peer to the ***%WIN32DMIPATH%*** subdirectory.

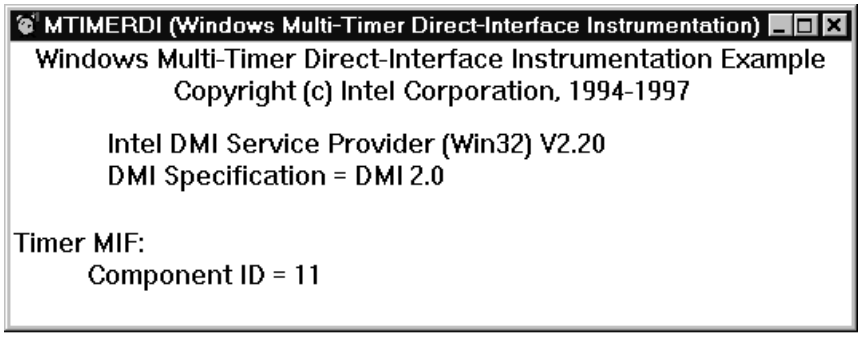


If the MTIMERDI instrumentation cannot register with the local DMI-SP, it displays this window.

If the MTIMERDI instrumentation is not resident, the MTIMERMA program displays this window.



The first thing MTIMERMA does is set up the Windows environment by providing a main entry point for the management application and registering the Multi-Timer window class. Then it handles the Windows-specific actions, such as creating, destroying and repainting the dialog.



If the MTIMERDI instrumentation is resident, then the Multi-Timer component MIF data is ready to be managed.

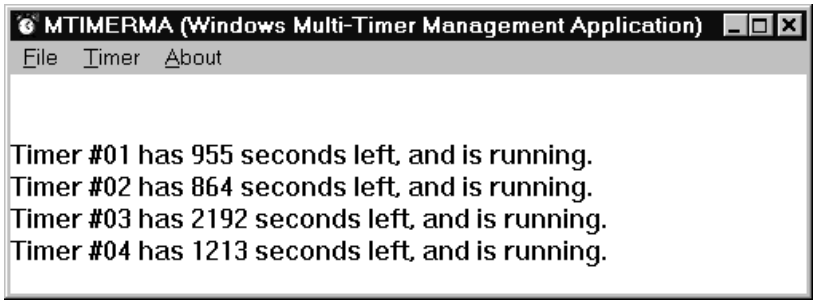
Next, the application checks that the Service Provider is loaded and the Multi-Timer component MIF file has been installed into the component MIF database. It also checks that it has registered with the Service Provider; if it has not, it performs the registration, initializing several variables at that time. (*Remember: Any management application that wants to receive indications must register with the Service Provider.)*

MTIMERMA then retrieves the ComponentID of the MTIMERDI component. Because this is assigned by the Service Provider and cannot be known ahead of time, it must be retrieved dynamically by listing all the components and filtering for the correct one.

After the preparation is complete, the MTIMERMA application sample is ready to do the work of defining, running and manipulating the Timers, based on input from the user dialog box (setting, stopping, restarting or deleting a Timer).

The main window of the MTIMERMA program is empty when it first displays, and there are no Timers defined the first time the component MIF data is used. The FILE menu has one option, Exit. The ABOUT menu brings up version-related information about the

Multi-Timer Management Application. The TIMER menu has the following options: Edit, Add, Delete, Start All, Stop All and List.



Use the various TIMER menu options to manipulate the Timers.

When a Timer counts down to zero or is deleted, an indication is sent to the management application, which then displays a message box to the user.

The List option under the TIMER menu retrieves attribute values for all Timers active at that instant. This display is static until (1) the user turns the List feature off, then turns it back on again; or (2) a Timer expires, at which time the values for all active Timers are updated within the MTIMERMA display.

The MTIMERDI instrumentation counts down each Timer every time the system clock ticks off one second. Whenever a Timer reaches zero, the MTIMERDI instrumentation sends an indication of this event to the Service Provider, which informs MTIMERMA that the Timer has expired.

At this time, a pop-up message window alerts the user and identifies the ID number of the Timer that has expired.

Once a Timer has gone off, it shuts itself down until the user sets it again.

Finally, when you close MTIMERMA, it unregisters with the Service Provider.

Exercising the Multi-Timer DI instrumentation

After the component instrumentation is successfully registered with the local DMI Service Provider, any management application can make requests to the DMI-SP to get attribute values in the Multi-Timer's groups.

When the DMI-SP receives a request about instrumented attributes in the Multi-Timer component, the Service Provider will call one of the MTIMERDI.EXE entry points—*GetAttribute(), SetAttribute(), GetNextAttribute(), AddRow(), DeleteRow()*—to service the request. Use the MTIMERMA application to define one or more Timers. These Timers generate indications whenever they count down to zero or are deleted.

You can use DCTS2 to connect to the local DMI-SP and activate its Event Monitor window to watch for Multi-Timer-related indications being sent out by the DMI-SP whenever a Timer elapses. Notice that you can enable multiple management applications to monitor the system for

indications for the same set of components (both DCTS2 and the MTIMERMA application are managing the MTIMERDI component).

Note: If a management application attempts to access an instrumented attribute without the instrumentation being registered with the DMI-SP, the operation fails in the following manner: The Service Provider returns the "Direct Interface Not Registered" message.

After the component is successfully registered with the local DMI Service Provider, your management application can send requests to the DMI-SP to get attribute values in different database groups. When the DMI-SP receives a request, it will call one of the MTIMERDI.EXE entry points—*GetAttribute(), SetAttribute(), GetNextAttribute(), AddRow(), DeleteRow()*—to service the request.

How to exercise the Multi-Timer sample with DCTS2

1. Verify that the WIN32SL.EXE service is loaded and running.
2. Copy the MTIMERMA.EXE, MTIMERDI.EXE and MTIMERDI.MIF files to the same directory.
3. Run the MTIMERDI.EXE instrumentation installation program.
4. Run the DCTS2 program. Create a connection to your local system. Open a Browser window for the local system connection's DMI Service Provider database. Open the Event Monitor window.
5. Expand the "Windows Multi-Timer Example" component (double-click the tree node box labeled with a C, located to the left of the component's name). Expand the Multi-Timer Control and Multi-Timer Table groups.
6. You can add a Timer by setting the Value of the Starting Ticks attribute in the Multi-Timer Control group. This step adds an instance to the Multi-Timer Table group and initiates a counter for the new Timer, which starts counting down immediately within the instrumentation.

   When the counter counts down to zero, it generates an indication which you can see in the DCTS2 Event Monitor window. Use a small value for the Timer to keep the waiting interval short.
7. You can stop, restart or delete a Timer by setting the Value of the Timer Control attribute in the group, Multi-Timer Table. Select this attribute in the DCTS2 Browser window and drop down the Value field's selection list. You will find three values enumerated: Stop, Run and Deleted. Stop causes the instrumentation to stop decrementing the Timer counter. Run causes the instrumentation to resume decrementing the timer counter and updating the component database in the DMI-SP. Delete causes the instrumentation to remove that instance of the Timer. ***When the instrumentation receives a request to set the Timer Control attribute to Delete, it removes that instance of a Timer from the Multi-Timer Table group and generates an indication, which you should see in the DCTS2 Event Monitor window.***
8. Examine other attributes, such as the Timer Count, New Timer Number, Timer Number, Event Type, and so on.

Exiting the Multi-Timer sample

When you choose to exit the MTIMERMA application, it first cancels its indication subscription and then unregisters with the Service Provider; it then terminates. The Multi-Timer DI Instrumentation remains resident until its installation window is closed. To remove the Multi-Timer Component from the DMI MIF database, use DCTS2.

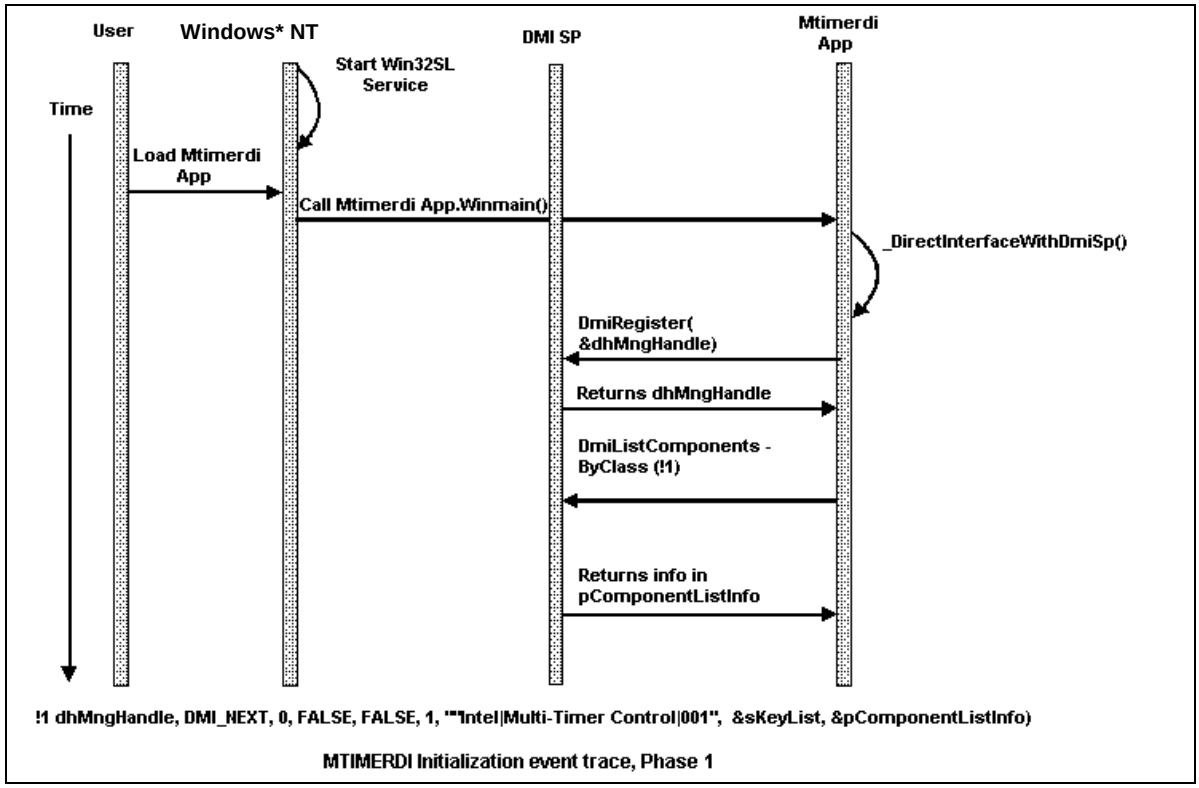
Multi-Timer event diagrams

The figure below shows the events that take place at various stages of the program initialization, operation and termination.

Multi-Timer DI Instrumentation initialization, Phase 1

The following sequence of operations takes place in Phase 1 of initialization:

1. During the system initialization, the operating system loads and starts the Win32sl.exe service.
2. The user runs the mtimerdi.exe instrumentation.
3. The operating system loads and runs the MTIMERDI.EXE instrumentation.
4. The MTIMERDI instrumentation calls its private *_DirectInterfaceWithDmiSp()* function to register with the Service Provider. This function makes calls to other functions to perform its operation, which correspond to the events that follow.



MTIMERDI Initialization event trace, Phase 1

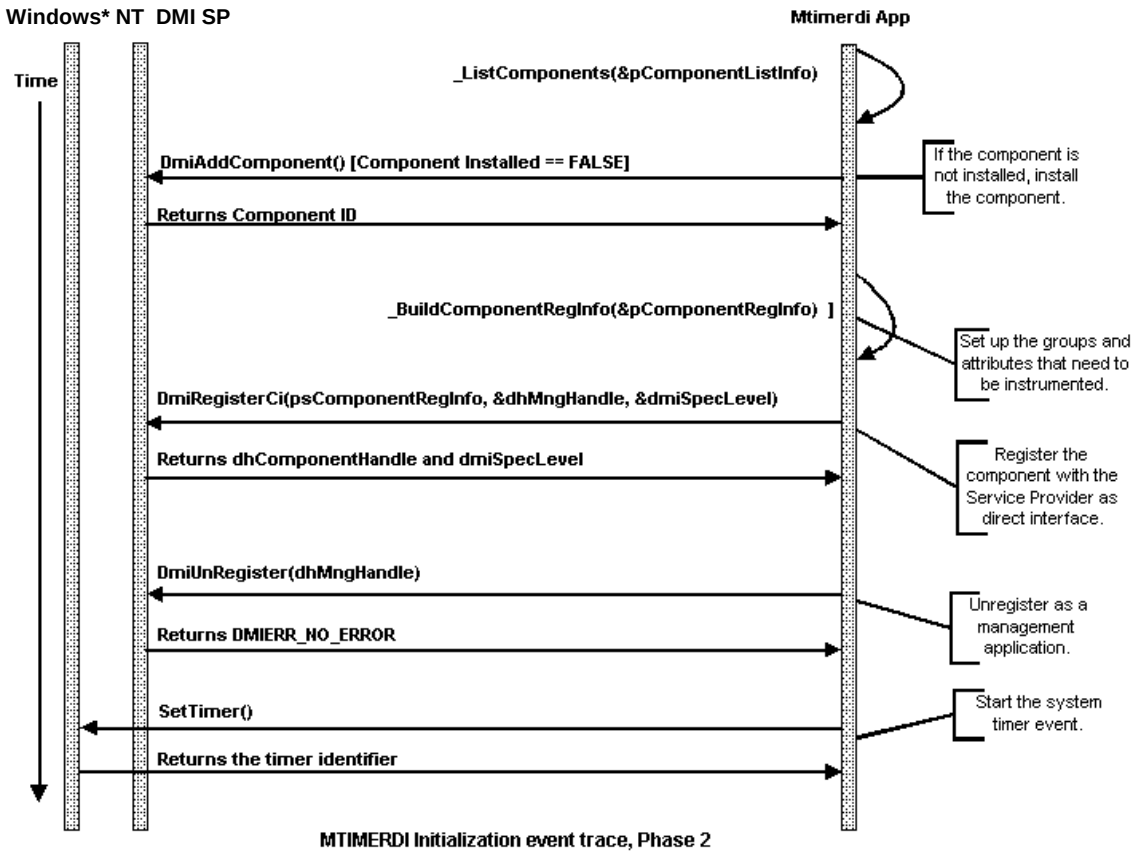
5. The MTIMERDI.EXE instrumentation registers with the Service Provider as a management application.
6. The Service Provider returns a valid management handle to MTIMERDI.EXE.

7. Using this management handle, the MTIMERDI instrumentation calls the *DmiListComponentsByClass()* function, querying the DMI-SP to see if the MTIMERDI component has been installed by passing the class string, Intel|Multi-Timer Control|100 as a filter.
8. The DMI-SP returns the query information to MTIMERDI.

Multi-Timer DI Instrumentation initialization, Phase 2

The following sequence of operations takes place in Phase 2 of initialization:

1. The MTIMERDI instrumentation calls its private *_ListComponents()* function to determine whether the MTIMERDI component has been installed.
2. If the component is not installed, MTIMERDI requests the Service Provider to install the component.
3. The Service Provider installs the component and returns success status.
4. MTIMERDI calls its private *_BuildComponentRegInfo()* function to set up the groups and attributes in the MTIMERDI.MIF file that need to be instrumented.



MTIMERDI Initialization event trace, Phase 2

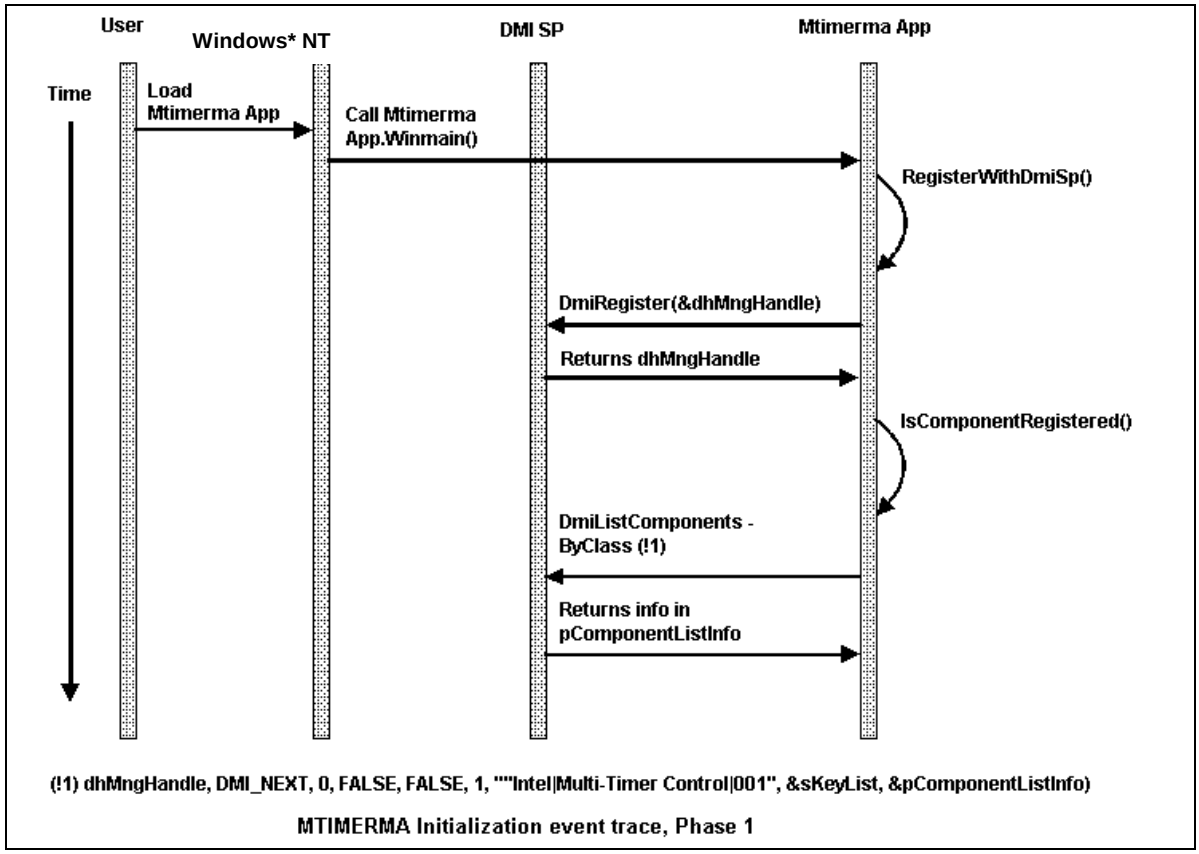
5. MTIMERDI instrumentation requests the Service Provider to register the component as direct interface by calling DMI-SP's *DmiRegisterCi()* function.
6. The Service Provider registers the component as direct interface. It then returns the component handle and the Service Provider's specification version to the MTIMERDI instrumentation.
7. The MTIMERDI instrumentation requests the DMI-SP to Unregister the application as a management application, by calling the DMI-SP's *DmiUnregister()* function.
8. DMI-SP unregisters the application and returns success status.
9. MTIMERDI instrumentation calls the operating system *SetTimer()* function to create a system timer.
10. The operating system returns the timer identifier to MTIMERDI instrumentation.

After these steps, the MTIMERDI instrumentation is ready to receive requests from the MTIMERMA.EXE management application; you can invoke MTIMERMA.EXE now.

Multi-Timer Management Application initialization, Phase 1

The following sequence of operations takes place in Phase 1 of initialization:

1. The user runs the MTIMERMA.EXE application.
2. The operating system loads and runs the MTIMERMA.EXE application.



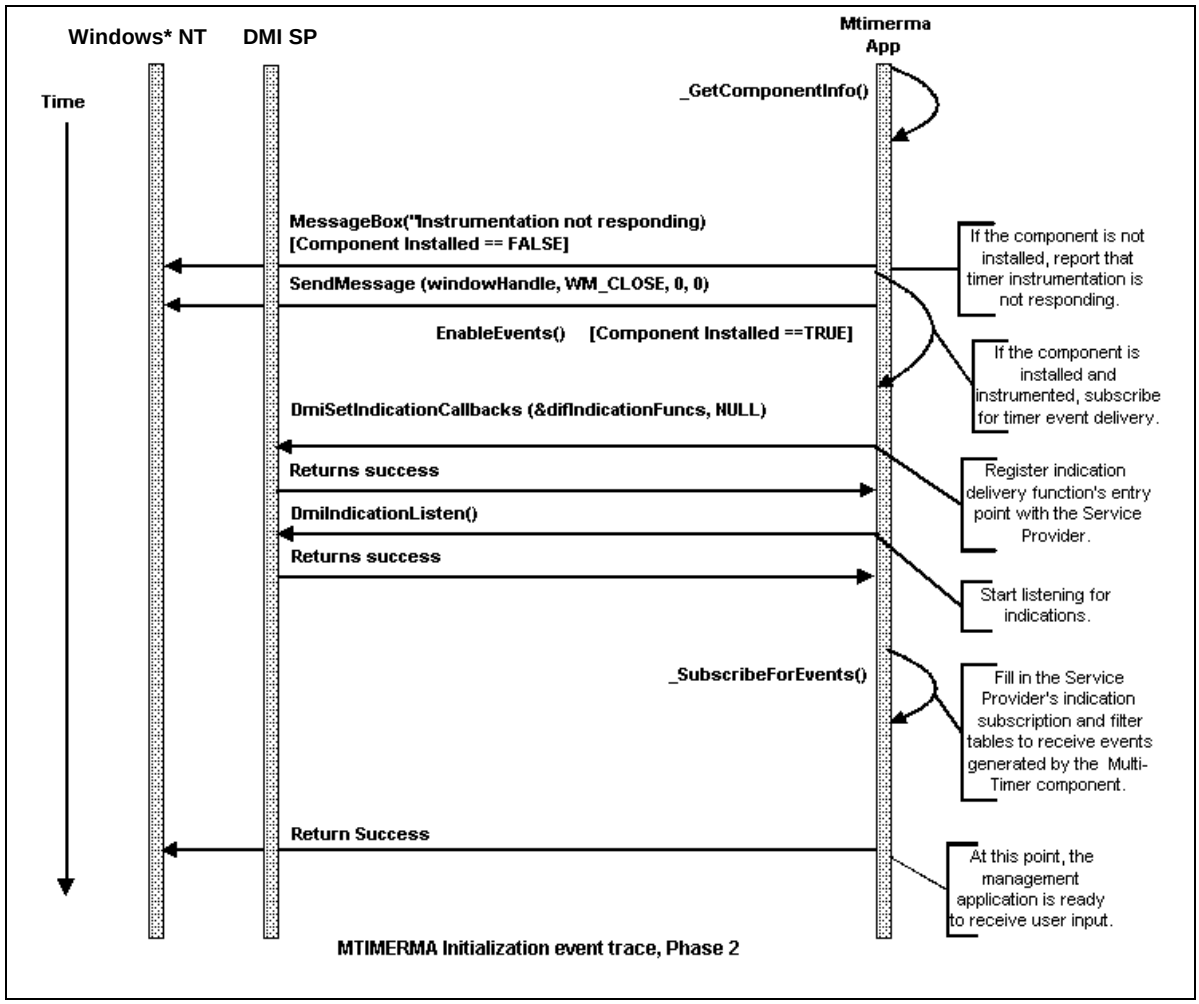
MTIMERMA Initialization event trace, Phase 1

3. The MTIMERMA.EXE application calls its *RegisterWithDmiSp()* function to register with the Service Provider. This function makes calls to other functions to perform its operation, which correspond to the events that follow.
4. MTIMERMA.EXE calls its *IsComponentRegistered()* function, which checks to see whether the MTIMERDI component is installed and registered with the Service Provider.
5. The *IsComponentRegistered()* function call results in a call to the *DmiListComponentsByClass()* function, with "Intel|Multi-Timer Control|001" as the class filter.
6. The Service Provider returns the component list information.
7. If the MTIMERDI component is not registered with the Service Provider, the MTIMERMA.EXE application displays a message indicating that the instrumentation is not responding.
8. If it is determined that the component is not installed in the above step, MTIMERMA.EXE will terminate.
9. The Service Provider returns a valid management handle to MTIMERMA.EXE.

Multi-Timer Management Application initialization, Phase 2

The following sequence of operations takes place in Phase 2 of initialization:

1. MTIMERMA.EXE calls its private *_GetComponentInfo()* function to get the ComponentID.



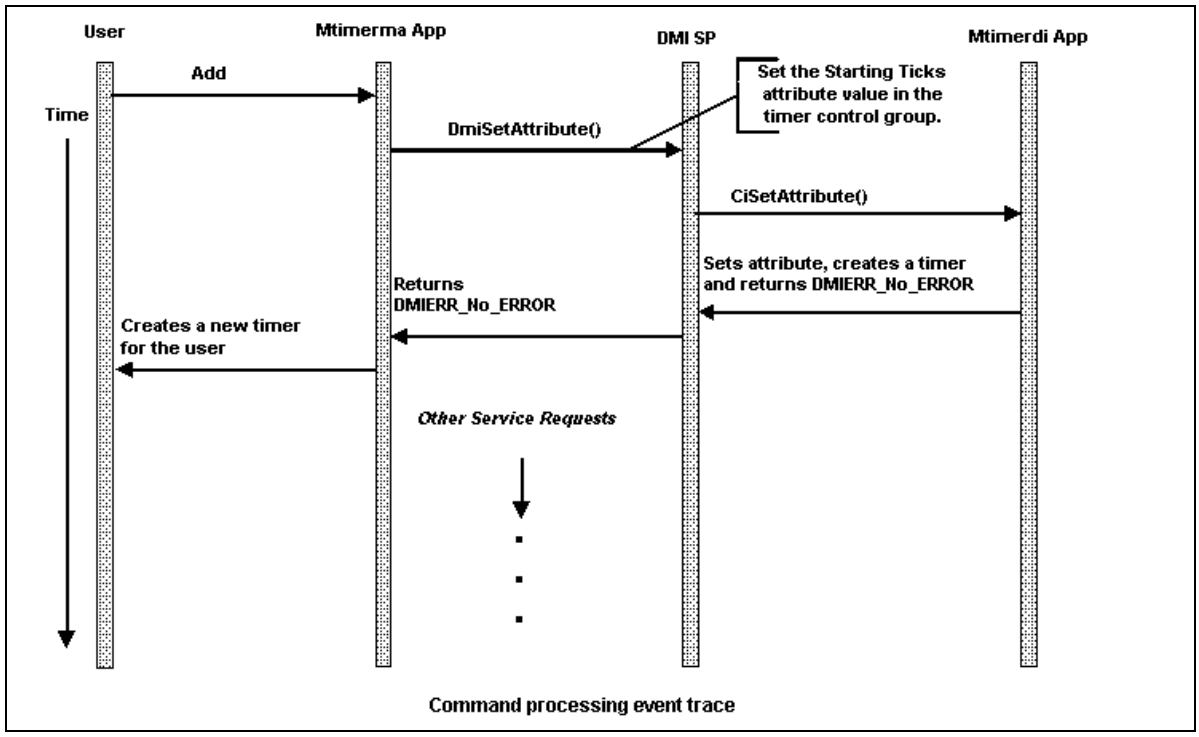
2. If the MTIMERDI component is registered with the Service Provider, the MTIMERMA.EXE calls its *EnableEvents()* function.
3. MTIMERMA.EXE registers its indication delivery function entry points with the DMI-SP by calling the Service Provider function *DmiSetIndicationCallBacks()*.
4. The Service Provider returns success.
5. MTIMERMA.EXE asks the DMI Client front end to start listening for indications by calling the *DmiIndicationListen()* function.
6. The Client front end returns success.
7. To enable indication deliveries by the Service Provider, the MTIMERMA.EXE application fills in the Service Provider's indication subscription and filter tables with Multi-Timer event information; this is done by calling its private function, *_SubscribeForEvents()*.
8. The MTIMERMA.EXE application returns success to the operating system.

At this point, both the MTIMERDI instrumentation and MTIMERMA management applications are initialized and ready to communicate. In this phase, the user can make requests to the instrumentation via the timer interface provided by the management application.

Command processing

The following sequence of operations takes place during command processing:

1. The user selects the Add item from the Timer menu in the management application.

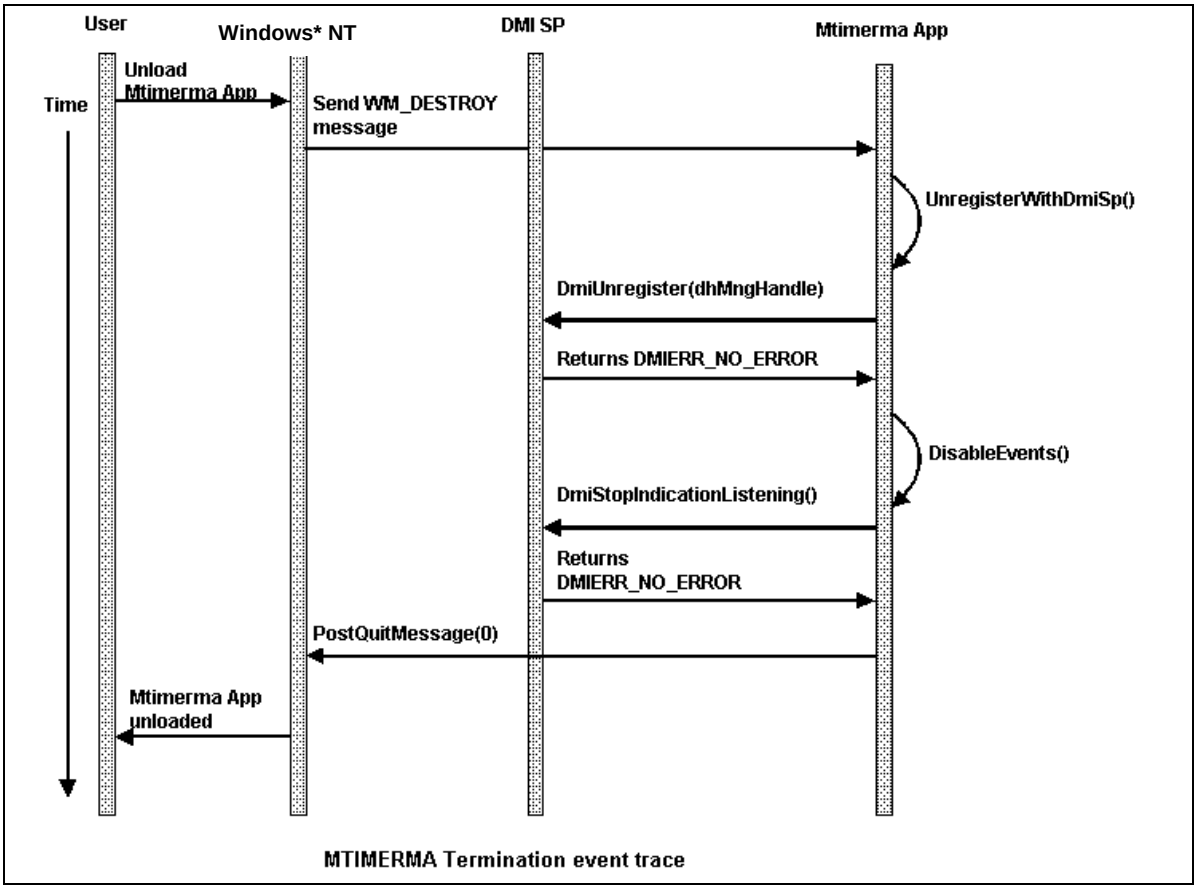


2. The MTIMERMA management application makes a *DmiSetAttribute()* request to the Service Provider, passing a value for the Starting Ticks attribute in the Multi-Timer Control group.
3. The Service Provider passes the information to the MTIMERDI instrumentation by calling its *CiSetAttribute()* entry point.
4. The MTIMERDI instrumentation writes the requested value to the Starting Ticks attribute in the Multi-Timer Control group, then creates a timer. The number of ticks in the Timer's initial setting will be the value of the Starting Ticks.
5. MTIMERDI instrumentation returns success to the Service Provider.
6. Service Provider returns success to the management application.
7. The management application displays the new timer number and its remaining ticks to the user, provided that the List item has been selected from the Timer menu.
8. The user can perform other timer operations, such as Start All, Stop All, Delete, and so on, by selecting them from the Timer menu in the MTIMERMA.EXE management application.

Terminating the management application

The following sequence of operations takes place during termination process:

1. The user requests the operating system to unload the MTIMERMA application by selecting Exit from the FILE menu.



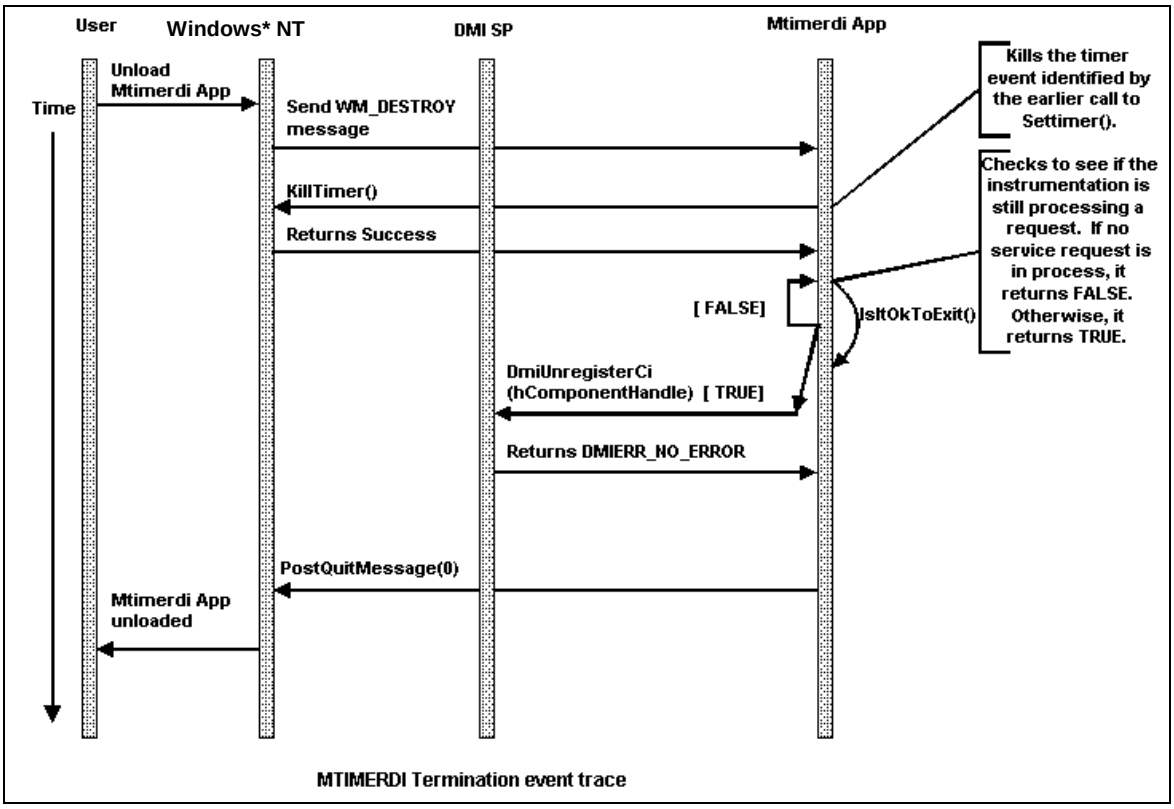
MTIMERMA Termination event trace

2. The operating system sends the WM_DESTROY message to MTIMERMA instrumentation.
3. MTIMERMA calls its *UnregisterWithDmiSp()* function to unregister as a management application. The call to this function triggers the events that follow.
4. MTIMERMA calls *DmiUnregister()* to unregister as a management application.
5. The Service Provider returns success to MTIMERMA.
6. MTIMERMA calls its *DisableEvent()* function, which calls the Service Provider's *DmiStopIndicationListening()* function to stop indication deliveries.
7. The Service Provider acknowledges the request and returns success.
8. MTIMERMA sends a *PostQuitMessage()* to the operating system.
9. The operating system unloads the MTIMERMA management application from memory.

Terminating the instrumentation and its application

The following sequence of operations takes place during termination process:

1. The user requests the operating system to unload the MTIMERDI instrumentation by selecting Exit from the FILE menu.



2. The operating system sends the WM_DESTROY message to MTIMERDI instrumentation.
3. MTIMERDI instrumentation calls the operating system function *KillTimer()* to abort the timer event that was created by the earlier call to the *SetTimer()* function.
4. The operating system returns success.
5. MTIMERDI.EXE calls its *IsItOkToExit()* function to see if the instrumentation is still processing a request. If no services are in process, it returns FALSE.
6. If the above step returns FALSE, MTIMERDI unregisters as direct interface by calling the Service Provider's *DmiUnregisterCi()* function.
7. The Service Provider unregisters the instrumentation and returns success.
8. MTIMERDI sends a *PostQuitMessage()* to the operating system.
9. The operating system unloads the MTIMERDI instrumentation from memory.

Troubleshooting runtime problems

To debug the Multi-Timer Management Application, you can execute the MTIMERDI.EXE and MTIMERMA.EXE programs under the Microsoft Visual C++ 4.2 development environ-

ment and set breakpoints on the five instrumentation functions defined in the MTIMERMA.C module. Use the MTIMERMA program to access the instrumented attributes.

Stepping through the Multi-Timer sample with a debugger

To debug the MTIMERMA management application and the MTIMERDI instrumentation application, perform these steps:

1. Run the Microsoft Visual C++ 4.2 Development Studio program.
2. Load the MTIMERMA.MDP workspace.
3. Open the MTIMERDI.C module and set breakpoints on the five instrumentation entry points: *GetAttribute(), SetAttribute(), GetNextAttribute(), AddRow(), DeleteRow().*
4. Run the MTIMERMA application.
5. After invoking DCTS2, connect to the local system and open a Browser into the local system's Service Provider database. Expand the component tree, then click to view the attribute values in different groups and tables in the MTIMERDI.MIF component MIF file.

Note: **You'll need to copy the MTIMERDI.MIF file into the same directory that the Development Studio's debugger copied the MTIMERDI.EXE file.** When you use one of the browsing tools (either DCTS2 or the MTIMERMA application) to access attribute values, the Service Provider will call the entry points to the MTIMERDI instrumentation and break on the breakpoints that you set in Step 3 above.

<This page is intentionally blank.>

Chapter 6 - DMI 2.0 and SDK Notes

Overview

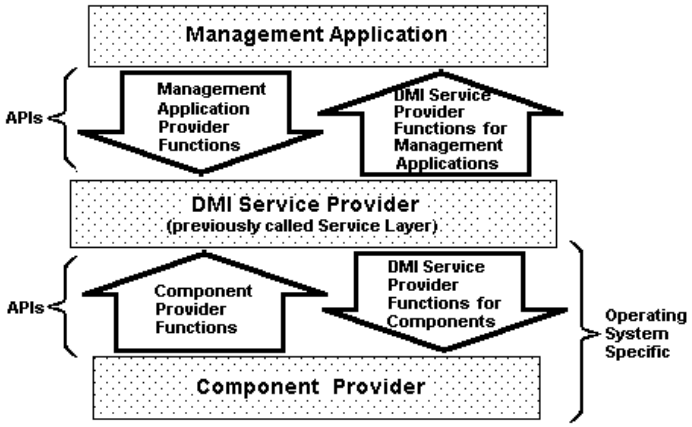
This chapter details the entities and operations defined for the DMI 2.0, and provides specifics about how the binaries provided with this SDK have been implemented to support the requirements outlined in *DMI 2.0 Specification*.

Special notes about the DMI v2.0

Interfaces to DMI-SPs

All interactions through the procedural **Management Interface** (MI) and **Component Interface** (CI) occur as Remote Procedure Calls (RPCs) to the DMI 2.0 procedural interfaces (APIs). The *DMI Specification* identifies four broad abstract API classes within the DMI framework:

1. DMI Service Provider Functions for Management Applications
2. Management Application Provider Functions
3. DMI Service Provider Functions for Components
4. Component Provider Functions



Abstract classes of DMI 2.0 procedural APIs

Note: These four classifications are used within the *DMI Specification version 2.0* to organize its presentation of DMI procedures. The *actual* set of APIs provided for the DMI entities is implementation-specific.

In the diagram above, each API arrow is directed *away* from the DMI entity that must provide that API's entry points and towards the DMI entity that will call those entry points. Each API entry point provider must define procedures corresponding to those functions defined for that API.

In addition to the required functions defined for the **Management Interface,** the *DMI Specification* also describes some MI extensions it refers to as the optional MI support functions, to be provided as implementation-specific extensions to the MI. The functions defined for the **Component Interface** are operating system-specific.

Within the limits of its implementation[13], the DMI 2.0 Service Provider must also support DMI 1.x legacy MI code. Since the block interface was supported by DMI 1.x Service Providers as *local interface*, no RPC support layer is required. The DMI-SP controls communication between itself and DMIv1.x management applications using a **Data Block Management Interface**. The DMI-SP controls communication between itself and v1.x manageable products using a **Data Block Component Interface**.

Block interface for MI and CI functions: DMIAPI32.DLL

Management applications and direct interface component instrumentation that need to use the block interface instead for its MI or CI functions must <include> the header file DMIAPI.H, which links into the DMIAPI.LIB file. The commands supported by the block API of this SDK's DMI-SP are identical to those entry points and command codes that are described in the *DMI 1.1 Specification.* Therefore, this *Reference* will not reiterate that information.

Supporting DMI 1.1 Events

The Intel DMI 2.0 Service Provider is compatible with both DMI 1.1 and DMI 2.0 component instrumentation. As part of this compatibility, the Service Provider will map DMI 1.1 events to DMI 2.0 indications. To effect this mapping, the Service Provider requires DMI 1.1 events to be in the correct format, as described in section 5.10 of the *DMI 1.1 Specification* and subsequent errata.

The correct format for the DmiEventData command block is:

Offset	Name	Description
0	iClassCount	number of group class data blocks
4	DmiClassData[]	list of group class data blocks

The correct format for the DmiClassData block is:

Offset	Name	Description
0	iComponentId	ID of the component for this event
4	osClassString	offset to the event group's class string
8	iRowCount	number of DMI row list offsets
12	oDmiGetRowList	offset to an array of offsets, where each offset is a reference to a DmiGetRowCnf structure.

[13] For example, the DMI-SP provided in this SDK supports only 32-bit code.

Properties of DMI-SPs

To identify a particular DMI-SP interface when more than one remoteable DMI-SP is connected, the management application builds a discovery list of available machines (system nodes visible to the local system) and establishes a connection to a particular machine's DMI-SP. To make a list of available systems, the applications must utilize a low-level *native*[14] discovery mechanism.

The management application selects one system node at a time and uses a suitable RPC support layer function to attempt to submit a DMI *Registration* request to a DMI-SP on that system node. If the status code returned by the *Registration* function indicates success, then the management application is now registered with a DMI-SP on the target system node. Otherwise, the *Registration* request fails, which means there is no DMI-SP resident on that system node at the time the DMI *Registration* request was submitted. **Refer to Section 4.1 of the *DMI Specification* for details about binding to a managed machine.**

The DMI 2.0 procedural API supports the *DmiGetVersion()* function, which returns the DMI Specification version string in the second parameter, and the OS-Specific DMI Service Provider version in the third parameter. For the DMI Service Provider provided in this SDK, the function returns "Intel DMI Service Provider (Win32) V2.xx"[15] and "DMI 2.0" in the second and third parameters, respectively.

The DMI 2.0 procedural API also supports the system-level language functions, *DmiSet-Config()* and *DmiGetConfig()*, to enable management application providers to configure the default language setting. When no valid language designation applies to data to be returned, the DMI-SP will substitute data matching the default language setting.

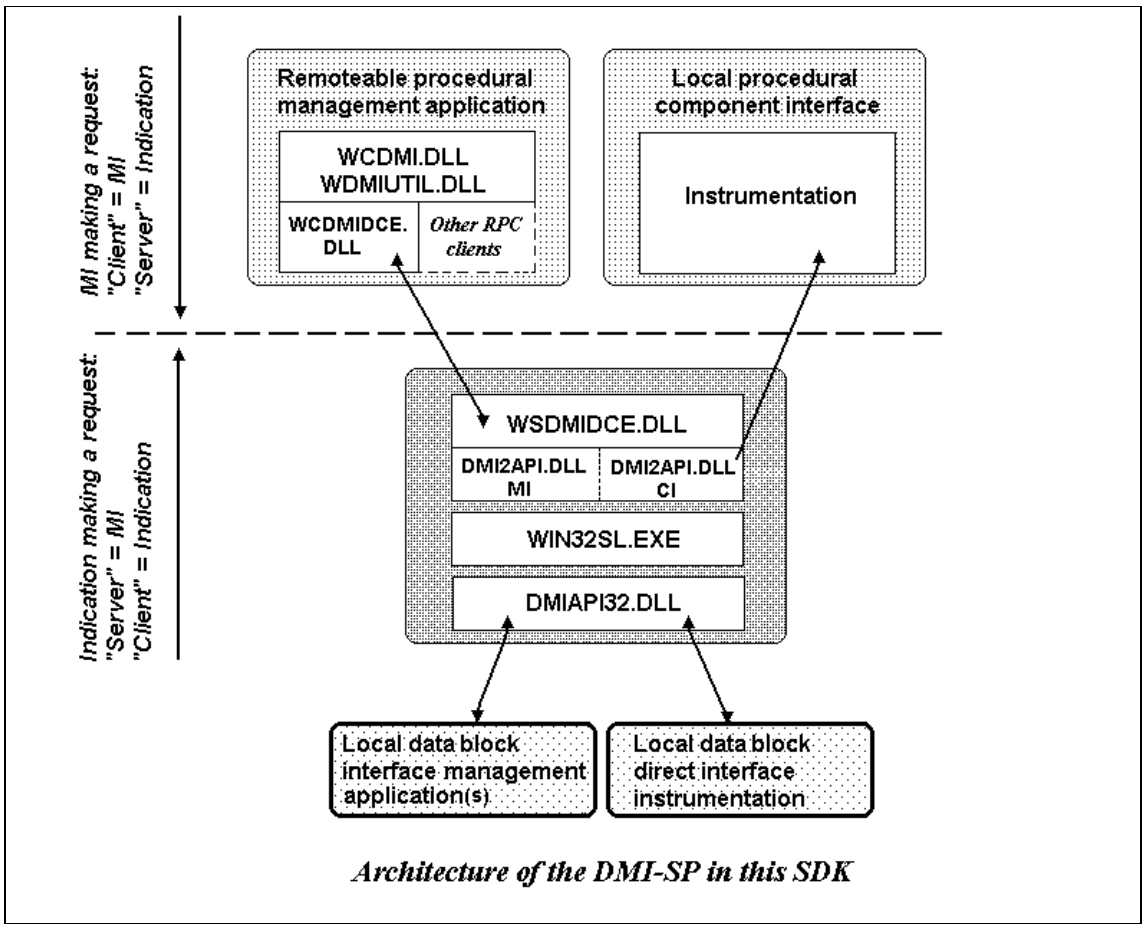
Architecture of the DMI for this SDK

The SDK's DMI-SP internal structure

The DMI-SP controls communication between itself and manageable products by means of the **Component Interface**. The DMI-SP in this SDK provides both a data block CI (DMIv1.x) and a procedural CI.

[14] Discovery is not supported by the DMI-SP provided with this SDK, nor by the SDK development libraries. Use the discovery mechanism available through your native operating system environment.

[15] In the version identifier "V2.xx, " the "xx" stands for the actual minor version number.



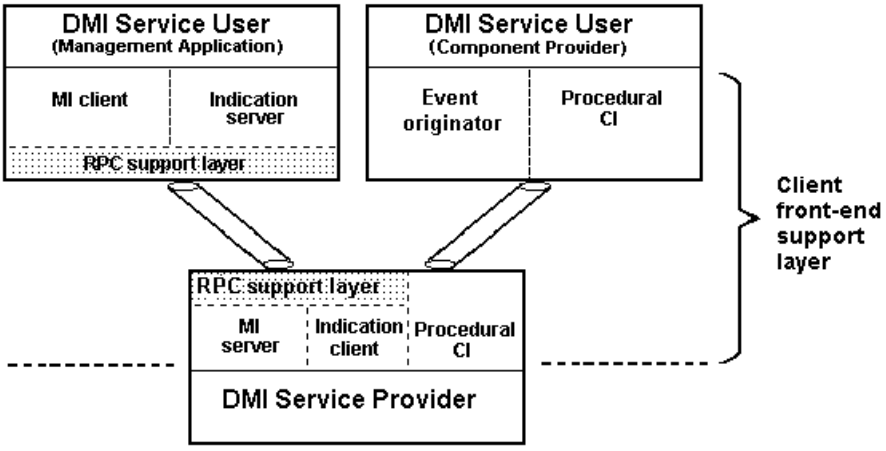
Architecture of the DMI-SP in this SDK

Client front-end interface

The DMI-SP provided in this SDK controls communication between itself and DMI 2.0 service users (management application providers). The functionality defined in the procedural MI, including the RPC support layer, is incorporated into each of the DMI entities. (See *The client front-end abstraction* in Chapter 7 in this *Reference*, and Section 1.7 in the *DMI Specification*.)

Communications between a ***DMI service provider*** (this DMI-SP) and a DMI management application (one type of ***DMI service user***) happen within an RPC client-server relationship within a **remote session** coordinated by **RPC support layers** on either side. The DMI service provider entity provides a **Management Interface server** and an **Indication client,** which are available to all management applications registered with the DMI-SP. The DMI service user entity provides a ***client front-end layer,*** which supports its **Management Interface client** and its **Indication server**.

Note: Connections between two RPC nodes can be made within the local system node entirely, or between a management application provider on a remote node and the local DMI-SP. The initial connection is made when the management application's RPC support layer subscribes to the DMI service provider's RPC support layer.



Client front end of this DMI-SP's remoteable interface

The management application provider first requests a **remote session** connection from the RPC support layer interface. Once connected, the associated client-server entities communicate with each other transparently by procedure calls across the RPC transport.

By sending command requests to the DMI-SP within the remote session, any management application can manipulate the MIF data of components. For fully instrumented MIFs, this means any DMI management application in an RPC node can act as a console to change the behavior in any other RPC node running the DMI 2.0. For RPC nodes whose DMI-SP supports it, unsolicited event notification can be sent out to management applications that are registered for events.

Data flow in the DMI-SP

Data flow in the DMI-SP starts with a service request from either a management application or a component instrumentation application. To communicate with the Service Provider, the management applications and component instrumentation need to register with the Service Provider and obtain a valid handle. This handle is used for subsequent service requests from the Service Provider. When registering with the DMI-SP, the application must also specify whether or not it wants to receive notifications for events.

Progression of a management request

There is a full presentation of the progression of a management request in Chapter 5, *DMI 2.0 code samples*, which includes process diagrams.

Progression of an event

At least three entities participate in an event: an Event Generator, an Event Consumer and an Event Reporter. An Event Generator can be a hardware or software device that has undergone a change in state or in which a specified condition has occurred. An Event Consumer is an entity interested in being notified when a specified event occurs. Calling the Service Provider entry point *DmiOriginateEvent()* causes events to be reported.

The event progression process starts with an Event Consumer (that is, a management application) subscribing for event notifications (indications). During the indication subscription, the management application provides the entry point of its indication processing function to the Service Provider.

When an event generator generates an event, the Event Reporter causes the Service Provider to deliver the indication to the Event Consumer by calling the *DmiOriginateEvent()* function.

When the Service Provider receives a *DmiOriginateEvent()* request, it delivers the event data to the Event Consumer by calling one of its indication processing function entry points, that was specified during the subscription process. The Event Consumer can then process the indication information and take appropriate actions. For more details and examples of event indication subscription and delivery, refer to the Multi-Timer code samples in Chapter 5 of this *Reference*.

Follow these steps to subscribe for event indication, when using this DMI-SP:

1. The Event Consumer registers with the Service Provider:

    a) On a per-target basis, configure the *DmiNodeAddress_t* structure by defining the address of the target, the type of RPC to use and the transport to use. If the target uses local RPC access, these values are {“”, “local”, “”}.

    b) On a per target basis, connect to the target machine and make it the default node, then register with the DMI-SP on that node. The client front end in this SDK also supports the *DmiSetDefaultNode()* and *DmiRegister()* functions. To perform the functions in separate calls, invoke the *DmiSetDefault-NodeAddress()* function to define the target as the default node. Then, call the *DmiRegister()* function to get a handle for the target DMI-SP. The client front end also supports the *DmiRemoteRegister()* function, which is functionally equivalent to calling both *DmiSetDefaultNode()* and *DmiRegister()*. This allows the management application to perform the operations of both functions in a single call, while using implicit binding.

2. The management application initializes a *DmiIndicationFuncs_t* data structure with the entry points of its indication processing functions. (This is done only once, when each management application is invoked.)

3. The management application passes the *DmiIndicationFuncs_t* data structure to the Service Provider by calling the *DmiSetIndicationCallbacks()* function. (This is done only once, when each management application is invoked.)

4. The management application tells the client front end to "listen" for incoming indications intended for the application by calling the *DmiIndicationListen()* function. (This is done only once, when the management application is invoked.)

5. On a per-target basis, the management application subscribes to receive indications, using the handle passed back as a part of registering to identify the intended target.

a) The management application must submit a *DmiAddRow()* request against the Indication Subscription Group to inform the target DMI-SP that it wants Indication Reception (see Section 3.3.1 of the *DMI Specification*).

b) The management application must submit another *DmiAddRow()* request against the Service Provider Information Filter Group to inform the target DMI-SP what sort of indications it wants (see Section 3.3.2 of the *DMI Specification*).

Before you configure the Event Generator, familiarize yourself with Section 3.2.2 of the *DMI Specification*. Then, follow these steps when using this DMI-SP:

1. The Event Generator fills a *DmiMultiRowData_t* data structure with information about the indication. A *DmiMultiRowData_t* structure is a table (data array), consisting of one or more rows (members), where each row is a *DmiRowData_t* data structure. The minimum set of rows for each event in the *DmiMultiRowData_t* structure is two.

   a) The first row of the event data (that is, the *DmiMultiRowData_t* data array) contains information about the Event Generation Group[16] that is associated with the current event.

   b) The second row of the event data carries information about the group generating the current event (indication). The group generating the event can be *any* group within the component—either a value-added private group defined by the manufacturer or a group conforming to one of the DMTF-approved Standard Groups definitions.

2. After initializing the second row of the *DmiMultiRowData_t* array, the Event Originator calls *DmiOriginateEvent()*, which passes this event data array to the DMI-SP.

After completing these steps, the Service Provider delivers indication information to the management application. To stop receiving indications, the management application needs to call the *DmiStopIndicationListening()* function.

Using this DMI SDK

The SDK install program, by default, configures the DMI-SP for auto-start. However, if the DMI-SP is shut down for any reason, it is possible to restart it without rebooting the system, by either clicking the DMI 2.0 Service Provider icon or by starting the WIN32SL.EXE program from the *%WIN32DMIPATH%*\BIN subdirectory.

Note: If no environment variable specifies the DMI root directory, the DMI Service Provider will use C:\DMI\WIN32 by default. This SDK Setup program adds the variable *WIN32DMIPATH* to AUTOEXEC.BAT on Windows 95 systems. If the user accepts the Setup program's default directory paths on a new installation, then the value of *%WIN32DMIPATH%* is C:\DMI\WIN32. The SDK Setup copies its files into subdirectories of the path specified by *%WIN32DMIPATH%*.

[16] The Event Generation Group is defined as a Standard Group by the DMTF and is discussed in section 3.2.2 of the *DMI Specification*.

Operating system-specific notes

The DMI SDK runs on the Windows 95 and Windows NT platforms. Except for some installation differences described later in this chapter, the Service Provider's functionality and behavior are identical under both of these operating systems.

Windows 95-related notes

Under Windows 95, the Service Provider is registered as a service that starts when Windows boots up. This is located in the registry under the Win32sl key in the `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\RunServices` path.

To prevent the Service Provider from being invoked during the bootup phase, you can remove the Win32sl key from the Windows 95 registry.

To create an icon for the DMI Service Provider, without having the resident DMI-SP appear on the desktop, add the -i parameter to the command line defined in the shortcut's Target property.

Windows NT-related notes

Under Windows NT, the Service Provider is implemented as an NT Service. It is started during the operating system bootup sequence. You must have administrator privileges to install the SDK on the Windows NT system. Once the SDK is successfully installed on the system, anyone can use the Service Provider Services; administrator privileges are no longer needed.

Files included in this SDK

%WIN32DMIPATH%\BIN subdirectory

This directory contains the DMI-SP binary files.

WIN32SL.EXE	DMI 1.0 Service Provider
DMIAPI32.DLL	32-bit DMI API to the WIN32SL.EXE Service Provider
NTEVENT.DLL	Service Provider NT Event Log API
WCDMI.DLL	DMI 2.0 Client Front End (Generic RPC Client)
WCDMIDCE.DLL	DMI 2.0 Client (specific for DCE/RPC Client)
WDMI2API.DLL	DMI 2.0 Server Interface wrapper to DMI 1.0 API
WDMIUTIL.DLL	DMI 2.0 Client Helper Functions API
WSDMIDCE.DLL	DMI 2.0 DCE/RPC Server

%WIN32DMIPATH%\DOC subdirectory

This directory contains document files included in the DMI 2.0 SP SDK.

REFMAN.PDF	This *SDK Reference*
LICENSE.TXT	Text file outlining Intel's SDK licensing policies
README.TXT	Read Me text file
RELNOTES.TXT	Release notes text file
AUTHRPC.HTM	*Creating Secure Remote Access to DMI Information*

SECURE2.0.HTM	*DMI 2.0 Security Token Proposal*
REFERENCES.HTM	*DMTF Reference and Association Proposal*

%WIN32DMIPATH%\INCLUDE subdirectory

This directory contains the <include> files for management application and component instrumentation developers.

WCDMI.H	DMI 2.0 Procedural Interface (Generic RPC) header file
DMIAPI32.H	DMI 2.0 Data Block Interface header file
DMI2CI.H	DMI 2.0 Component Interface header file
DMI2COM.H	DMI 2.0 Types file, which contains information on data structure definitions and function prototypes used by the DMI Service Provider
DMI2ERR.H	DMI 2.0 Error codes header file
DMI2IND.H	DMI 2.0 Client App Indication Xface header file
DMI2MEM.H	DMI 2.0 Memory Management Helper Functions header
DMI2SRV.H	DMI 2.0 Management Interface header file

%WIN32DMIPATH%\LIB subdirectory

This directory contains library files for management application and component instrumentation developers.

WCDMI.LIB	DMI 2.0 Procedural Interface Library (WCDMI.H)
DMIAPI.LIB	DMI 2.0 Data Block Interface Library (WCDMI.H)

%WIN32DMIPATH%\MIFS\BACKUP subdirectory

This directory contains the initial Service Provider MIF. Whenever the DMI-SP installs another MIF file, it copies the MIF file to this directory for future reference. The SP also makes a backup copy of the MIF database file, SLDB.DMI, in this directory.

WIN32SL.MIF	Service Provider 2.xx MIF file
SLDB.DMI	Backup copy of the DMI-SP component MIF database file

%WIN32DMIPATH%\MIFDB subdirectory

This is the component MIF database home directory. The first time the DMI-SP loads, it builds the SLDB.DMI database file (for internal use only) and creates the ERRORS.LOG file. Thereafter, the DMI-SP adds or removes components from the database file and reports any installation errors to the log file.

SLDB.DMI	Working copy of the DMI-SP component MIF database file; created by the DMI-SP the first time it boots up.
ERRORS.LOG	Text log file of runtime database administration errors.

%WIN32DMIPATH%\EXAMPLES directory

This directory contains the *%WIN32DMIPATH%*\EXAMPLES\TRUSTDI subdirectory and the *%WIN32DMIPATH%*\EXAMPLES\MTIMER subdirectory. See Chapter 5 for details about the SDK's code samples and their file lists.

Files not included in this SDK

To develop instrumentation and application code for DMI, you will need general Windows header files and libraries, which are not provided in this SDK.

Implementation details

General notes

1. The DMI-SP shipped with this SDK maintains its component MIF data directly as a database file. This DMI-SP *does* support MIF format files written for DMIv1.x, but it *does not* support MIF database implementations built under DMIv1.x Service Layers. There is no migration path that moves data from DMI 1.x databases into DMI 2.0 databases; therefore, DMI 1.x component MIF databases must be rebuilt by installing each component MIF into the new DMI-SP.

2. This SDK provides libraries that support the DMI v1.x block-oriented interface.

3. The DMI-SP provided by this SDK supports 32-bit DMI v1.x management applications and direct interface instrumentation.

4. All management and component instrumentation applications must include the CLIDMI.H header file, and link with the WCDMI.LIB library module to utilize the DMI Service Provider's services. Always <include> the WINDOWS.H header before the CLIDMI.H header since the CLIDMI.H depends on definitions in WINDOWS.H.

5. For proper memory allocation and de-allocation operations between the applications and the Service Providers, applications must use the memory helper functions defined in the DMI2MEM.H header file supplied with this SDK. For examples of using memory allocation functions such as *DmiNewString(), DmiNewTimeStamp(),* and so on, refer to the Trusted Code samples in Chapter 5.

6. To clean up the Service Provider's database and remove the MIF data for all installed components (for example, to enhance performance when the database becomes large), follow the steps listed below for your operating system platform. When the WIN32SL.EXE is invoked after the system reboots, the component MIF database will be rebuilt automatically.

   On a Windows NT system:

   1. Log in as administrator.[17]
   2. From the Control Panel, run the Services application.
   3. Select the Win32SL Service from the list of services.
   4. Click STOP.
   5. From the *%WIN32DMIPATH%*\MIFS\BACKUP directory, remove the SLDB.DMI file.
   6. From the *%WIN32DMIPATH%*\MIFDB directory, remove the SLDB.DMI and ERRORS.LOG files.
   7. Start the Win32SL Service.

[17] To perform these steps, you must have administrator privileges.

On a Windows 95 system:

1. Close the Win32SL service.
2. From *%WIN32DMIPATH%*\MIFS\BACKUP directory, remove the SLDB.DMI file.
3. From *%WIN32DMIPATH%*\MIFDB directory, remove the SLDB.DMI and ERRORS.LOG files.
4. Start the Win32SL service.

Management Applications

To be portable on both Windows 95 and Windows NT platforms, management application code must connect to the local machine by populating the *DmiNodeAddress_t* structure with {“ ”, “local”, “ ”} before calling the appropriate DMI *Registration* function, as described earlier in the *Indication Subscription* section of this chapter.

Otherwise, the *DmiRegister()* may fail, depending on the user access rights and network configurations. For an example of using the *DmiSetDefaultNode()* function, refer to the Trusted Code samples in Chapter 5.

Component Providers

1. The DMI-SP *does not* support runtime overlay instrumentation. Any existing runtime overlay instrumentation must be re-engineered to be a direct interface.

2. Calls to instrumentation entry points operate within the context of the DMI-SP.

3. Before registering as direct instrumentation with the Service Provider using the *DmiRegisterCi()* function, the component instrumentation must allocate a *DmiAccessDataList_t* data structure of the appropriate size. If you specify a size larger than the number of groups to be instrumented, the behavior is undefined.

   The size member of this data structure directly corresponds to the number of *DmiAccessData* structures used to describe the attributes that are to be registered for direct instrumentation. For example:

   ⇒ If ***all*** the attributes within a particular ***component*** are instrumented, the value of both the GroupID and the AttributeID are 0 (zero); only ***one*** *DmiAccessData* structure is used to describe the ***entire*** *DmiAccessDataList_t* structure.

   ⇒ If ***all*** the attributes within a particular ***group*** are instrumented, the value of the GroupID specifies the group and the AttributeID is 0 (zero). Only one *DmiAccessData* structure is used to describe the entire group, which becomes ***one*** member of the *DmiAccessDataList_t* structure.[18]

   ⇒ If only ***some*** of the attributes within a particular group are instrumented, the value of the GroupID specifies the group, and the AttributeID specifies one of the instrumented attributes. This defines ***one*** *DmiAccessData* structure. There

[18] There may be more groups to be defined, which will increase the size of the *DmiAccessDataList_t* structure.

will be one DmiAccessData structure for each instrumented attribute within that group. If ***three*** attributes within the specified group, there will be ***three*** *DmiAccessData* structures added to the members of the *DmiAccessDataList_t* structure.

5. Component instrumentation can be created either as a stand-alone executable (EXE) or as a dynamic-link library (DLL) which is instantiated by an executable. For an example of an instrumentation DLL, see the Trusted Code samples; for an example of instrumentation as a stand-alone executable, see the Multi-Timer samples. Both are found in Chapter 5, *DMI 2.0 code samples*.

6. This SDK provides the DMI procedure library to make writing instrumentation code as easy as possible; it also provides examples of direct interface instrumentation code. The code samples are described in Chapter 5, the procedure libraries in Chapter 7.

Event Generators

When allocating memory for the *DmiMultiRowData_t* structure for indications, the event generators must make sure that they allocate the correct amount of memory for the Standard Event Generation Group and the group which generates indications.

For example, the first *DmiRowData_t* structure in the *DmiMultiRowData_t* array, which is required for event indications, should have space for seven attributes for the Event Generation Group. The second *DmiRowData_t* structure in the *DmiMultiRowData_t* array may allocate space for the number of attributes contained in the group which generates indications.

In the Multi-Timer indication example, the Multi-Timer Table group has four attributes. Therefore, the Multi-Timer instrumentation module allocates space for four attributes in the second row of the *DmiMultiRowData_t* structure. If memory is not allocated properly (that is, space for eight attributes is created instead of four in the above example), the DMI Service Provider will not deliver indications to the Event Consumers. For more details, refer to the Multi-Timer code samples in Chapter 5.

Indication Consumers

For proper delivery of indication data to multiple management applications on multiple systems, the management applications should use the *DmiGetSubscriptionAddress()* function during indication subscription. Otherwise, the Service Provider delivers indications only to management applications local to the system. For an example of using the *DmiGetSubscriptionAddress()* function, refer to the Multi-Timer example in Chapter 5.

Component Pre-installation

All MIF files located in the C:\DMI\WIN32\MIFS directory will be installed when the Service Provider is loaded. Successfully-installed MIF files will be moved to the C:\DMI\WIN32\MIFS\BACKUP directory.

Component instrumentation must determine its component ID before requesting services from the Service Provider. To determine its component ID, component instrumentation should register itself as a management application and search for itself.

Component Installation Errors

If you try to install an invalid MIF file, the Service Provider returns the `0x20f` error code.

During installation, the Service Provider creates a file in the C:\DMI\WIN32\MIFDB directory with the same prefix as the MIF file being installed and with the suffix "err." For example, if a MIF file called comp.mif fails to be installed, then a file called C:\DMI\WIN32\MIFDB\COMP.ERR is created.

The error file contains information about where the errors were found in the MIF file. The error message format is:

```
ERROR: errorNum (lineNum, columnNum)
```

or

```
WARNING: errorNum (lineNum, columnNum)
```

where:

errorNum
: is the error found, the descriptions of all error numbers are given in the table below.

LineNum
: is the number of the line in the MIF file where the error occurred.

columnNum
: is the column number within the *LineNum* where the error occurred.

Error number	Error description
1643	Unused templates in the component
1703	Illegal value in group body
2101	Illegal character in comment
2641	There is no groupId 1 in the component
2647	Missing language statement
2691	Class string in componentId group does not match convention
2741	Class string in componentId group does not match convention
3105	Unexpected end of comment
3106	Unexpected end of file in comment

Error number	Error description
3201	Illegal character in escape sequence
3202	Illegal digit in escape sequence
3203	Illegal value in escape sequence
3204	Missing digit in escape sequence
3301	Illegal character within literal
3306	Unexpected end of file in literal
3307	Unexpected new line in literal
3401	Illegal character in source stream
3417	Only one component is allowed per MIF file
3503	Illegal value encountered in attribute body
3509	Attribute name expected
3510	Attribute ID expected
3511	Attribute name or ID expected
3512	'=' in attribute body expected
3513	Literal in attribute body expected
3514	Integer in attribute body expected
3515	Statement in attribute body expected
3518	Overlay name for the attribute value expected
3519	Attribute value expected
3520	Attribute value has an illegal size
3522	Access statement missing from attribute body
3523	Type statement missing from attribute body
3524	Value statement missing from attribute body
3525	Size specifier missing for string attribute
3528	Value conflicts with existing database
3529	Duplicate statement encountered in attribute body
3537	Cannot assign value to Write-Only attribute

Error number	Error description
3553	Illegal value in enumeration body
3559	Name for this enumeration expected
3562	'=' in enumeration body expected
3563	Literal in enumeration body expected
3565	Statement in enumeration body expected
3569	Value in enumeration body expected
3571	This enumeration had no statements
3578	Value conflicts with existing database
3579	An enumeration of this name was already encountered
3580	This feature is not currently implemented for enumerations
3603	Illegal value in component body
3608	Component definition expected
3609	Component name expected
3611	Component name expected
3612	'=' in component body expected
3613	Literal in component body expected
3615	Statement in component body expected
3616	Block statement in component body expected
3626	This component had no recognizable groups
3629	Duplicate statement encountered in component body
3651	Adding a language with number of groups different from number of groups in the component
3652	Two groups in the component share the same class string, but differ in definition
3653	Conflicting groups or Illegal value in table body
3654	Adding an already existing language
3655	Number of MIFs (in DmiAddGroup) doesn't match number of

Error number	Error description
	languages in the component
3656	No match between languages of component and languages statements in the MIF (in DmiAddGroup)
3657	Component mapping for this current language already contains this groupId. 2 MIFs with the same language
3659	Table name expected
3661	Table class, ID, or name expected
3662	‘=’ in table definition expected
3663	Literal in table definition expected
3664	Integer in table definition expected
3665	Statement in table body expected
3668	Overlay name for a table data entry expected
3669	Table entry value expected
3670	Table string value exceeds size specified in attribute
3671	This table contained no data entries
3674	Additional values in table row expected
3678	There is already a group or table with this ID
3679	Duplicate statement encountered in table definition
3684	Template class name expected
3685	The number of values exceeds the number of attributes
3686	',' or '},' after a table data entry expected
3689	Tables must be associated with keyed groups
3695	Key is not unique
3696	There is no default value in the template
3703	Illegal value in group body
3709	Group name expected
3710	Group ID expected
3711	Group class, name, or ID expected

Error number	Error description
3712	'=' in group body expected
3713	Literal in group body expected
3714	Integer in group body expected
3715	Statement in group body expected
3716	Block statement in group body expected
3727	This group had no recognizable attributes
3728	Value conflicts with existing database
3729	Duplicate statement encountered in group body
3738	Class statement missing from group body
3753	Unknown attribute ID specified in key statement
3760	Attribute ID in key statement expected
3762	'=' in key statement expected
3774	Attribute ID in key statement expected
3778	Duplicate ID encountered in key statement
3803	Illegal value encountered in component paths body
3809	Name or environment definition expected
3811	Name definition expected
3812	'=' in component paths body expected
3813	Literal in component paths body expected
3815	Statement in component path body expected
3817	'End' expected
3819	Path literal or "Direct-Interface" expected
3821	Required definitions were missing from component path body
3824	This attribute has no instrumentation for the current OS environment
3828	Value conflicts with existing database
3829	Duplicate statement in component paths body

Error number	Error description
3840	Expecting: DOS, OS2, UNIX, WIN16, or WIN32
3878	Templates must have unique class strings
3892	Templates are required to be keyed
3929	2 group statements in DmiAddGroup or 2 component statements
4306	Unexpected end of file in literal
4632	Out of memory in component
4648	Language or territory error in language string
4649	Encoding error in language string
4681	Database error in table
4900	File Error
4958	Unicode error

Other implementation tips

The DMI Component Test System (DCTS) is a great tool for exercising component instrumentation code. It can be used to feed data into the component instrumentation and verify correct functionality. For more detail on using DCTS, see Chapter 8.

Chapter 7 - DMI SDK Procedure Libraries

Overview

This chapter contains information about the procedure libraries included with this SDK, WCDMI.DLL and DMIAPI32.DLL, and the proprietary data structures defined and used with this implementation of the DMI-SP. The two procedure libraries separate the API functions provided to DMI service users into the two halves of the DMI 2.0 dual interface. The *Procedure Libraries* section of this chapter is divided into the types of APIs. Each API's section has a table of its functions (listed alphabetically) and a command list, broken down by category. The *Implementation-specific data types* section describes those proprietary data types used only by this implementation of the DMI-SP.

Information available in the *DMI 2.0 Specification*:

- The Event Generator functions include functions described in Section 8.2, plus some proprietary memory handling functions.
- The Management Application Provider functions include the standard MI functions described in Section 7.1 and the MI-related functions in the RPC/DCE client front end (proprietary to this DMI-SP), as outlined for the optional MI functions in Section 9.
- The Component Provider functions include functions described in Section 8, plus some proprietary memory handling functions.

Note: This chapter describes in detail ***only*** those functions and data structures unique to this SDK's procedure libraries and DMI-SP implementation. For those functions whose implementation in this DMI-SP are *identical* to the description in the *DMI Specification*, the reader should look in the *Specification* for details pertinent to the function of interest.

Procedure libraries

The client front end supports all the mandatory functionality of the DMI 2.0 including both MI and CI interfaces. However, it does not provide an optional interface defined in Section 9 of the *DMI Specification*. Instead, it provides an alternative API, facilitating connection establishment/tear-down, indication server control and memory management. The following sections describe this API in detail. Refer to the *DMI 2.0 Specification* and errata documents for the DMI 2.0 core API definition. The procedural MI and CI support provided by this DMI-SP implementation is wrapped in a client front-end abstraction. Part of the client front-end interface includes a layer of RPC/DCE support.

The block interface exposes the entry point *DmiInvoke(),* with which the DMI service user may pass a *command code* as part of each request block. Since the entry points and command codes for this interface are fully described by the *DMI 1.1 Specification,* refer to that *Specification* for detail on how the block interface is used.

The DMI procedure libraries described in this *Reference* are provided in two APIs: WCDMI.DLL and DMIAPI32.DLL. The WCDMI.DLL provides the *procedural* interface, while the DMIAPI32.DLL provides the *data block* interface.

WCDMI.DLL is implemented as a client front end to the RPC/DCE support provided within the Windows 95 and Windows NT operating systems. It provides all of the functionality described in the *DMI Specification* for the procedural entry points to both the MI and CI.

Note: The combined procedural MI/CI API for the DMI-SP in this SDK is implemented as a client front-end interface so that it can use DCE-specific features and yet provide a simpler, more abstract interface to management applications, component providers and event generators.

Backward compatibility with DMI 1.x

Developers who have programmed using DMI level DMIv1.0 and DMIv1.1 will recognize DMIAPI32.DLL as the DMI block interface for both the MI and the CI. DMIAPI32.DLL is implemented in this SDK, and supports the full MI functionality and the direct interface CI functionality defined in *DMI 1.1 Specification.* However, when a DMI service user uses the block interface entry point, *DmiInvoke()*, to register with the DMI-SP provided in this SDK, the DMI level returned is DMI v2.0.

Procedural interface for MI and CI functions: WCDMI.DLL

Management applications and direct interface component instrumentation that use the procedural MI or CI interface should <INCLUDE> the header file CLIDMI.H, which links into the WCDMI.LIB file. This allows your code to make calls into the client front end (WCDMI.DLL) of the DMI 2.0 Service Provider provided in this SDK.

Remote registration functions

The *DmiRemoteRegister()* and *DmiRemoteUnregister()* functions described in this section allow DMI clients to establish and break down a session with the remote DMI Service Provider in an RPC-independent fashion. The *DmiRegister()* and *DmiUnregister()* functions defined in the *DMI 2.0 Specification* are also supported by the client front end: they provide implicit register/unregister functionality with a globally-specified Service Provider. The *DmiSetDefaultNode()* function allows clients to define and change the Service Provider address used by the implicit DMI register.

DmiRemoteRegister

```
DmiErrorStatus_t DMI_API DmiRemoteRegister (
   /* [out] */  DmiHandle_t*             handle,
   /* [in]   */ DmiNodeAddress_t *       node
);
```

Description

Management applications use the *DmiRemoteRegister* function to open a session with the remote Service Provider specified by *node = { address, rpc, transport}.* Possible

values for RPC types, transport protocols and address formats supported by the client front end are listed in the RPC Specific Information below.

You must issue the *DmiRemoteRegister* or *DmiRegister* functions before any other DMI command is sent to the specified Service Provider.

To open a session with the remote Service Provider, you must bind with the specified RPC server and then send the *DmiRegister* command to the Service Provider. The DMI Service Provider uses this function to initialize its internal state for subsequent calls made by the application.

Parameters

Input

node Remote Service Provider with which to register.

Output

handle Unique per-session handle that must be used in all subsequent DMI commands.

Return value

```
DMIERR_NO_ERROR
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
DMIERR_CFE_RPC_NOT_SUPPORTED
DMIERR_RPC_INVALID_ADDRESS
DMIERR_RPC_TRANSPORT_UNAVAILABLE
DMIERR_RPC_SERVER_UNAVAILABLE
DMIERR_RPC_LOW_RESOURCES
```

DmiRemoteUnregister

```
DmiErrorStatus_t DMI_API DmiRemoteUnregister (
   /* [in] */   DmiHandle_t              handle
);
```

Description

Management applications use the *DmiRemoteUnregister* function to close a session with the remote Service Provider. To specify a session, the application supplies a session handle received from the *DmiRemoteRegister* or *DmiRegister* functions. The *DmiRemoteUnregister or DmiUnregister* functions must be the last DMI command in a session.

To close a session with the remote Service Provider, the client software sends the *DmiUnregister* command to the Service Provider, which then performs end-of-session cleanup actions. The client subsequently unbinds itself from DMI server.

Input Parameters

handle Session handle returned by the *DmiRemoteRegister* or *DmiRegister* functions.

Return value

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
DMIERR_RPC_LOW_RESOURCES
```

DmiSetDefaultNode

```
DmiErrorStatus_t DMI_API DmiSetDefaultNode (
   /* [in]   */  DmiNodeAddress_t *           node,
   /* [out] */  DmiNodeAddress_t * *          oldNode
);
```

Description

The *DmiSetDefaultNode()*function allows a management application to set and get the default Service Provider node. This default node serves as an implicit parameter in the *DmiRegister* function which opens session with the default Service Provider. The value set by this function is a global, per-process variable. The initial value of the default node addresses is the local SP via DCE RPC - *{"" ,"dce", "ncalrpc"}.*

This function is executed entirely by the client and does not send any command to the Service Provider.

Parameters

Input

node Pointer to node address to be set as default or NULL. If this parameter is NULL, the function does not change the current setting of the default node address.

Output

oldNode NULL, or a pointer to a memory block where the address of the allocated *DmiNodeAddress_t* structure specifying the previous default node settings will be returned. The client front end performs all the necessary allocations. The application must free the memory allocated for this output parameter.

If this parameter is NULL, the previous value of the node address is not returned.

Return value

```
DMIERR_NO_ERROR
DMIERR_OUT_OF_MEMORY
```

DmiRegister

```
DmiErrorStatus_t DMI_API DmiRegister (
    /* [out ] */  DmiHandle_t*            handle
);
```

Description

This function is defined in the DMI 2.0 specification, but has slightly different functionality in the client front end: *DmiRegister* opens a session with the default Service Provider that was last set by *DmiSetDefaultNode*. If there was no *DmiSetDefaultNode* function issued, the *DmiRegister* function opens a session with the *{"" ,"dce", "ncalrpc"}* node.

***Note*:** Using "*ncalrpc*" or "*ncacn_np*" transports on Windows NT imposes a security check when binding to the DCE server. This may cause a problem if a non-system application tries to register with the Service Provider running on the system account. To guarantee that the local SP is accessible to the application, use the *{"","local","","}* node address. This is particularly recommended when accessing the local SP for CI applications that need the MI interface.

The *DmiRegister* function, like the *DmiRemoteRegister* function, receives the node parameter implicitly. This allows an application to bind itself with the remote server either implicitly or automatically (similar to DCE RPC supported binding methods).

Output Parameters

handle Unique, per-session handle that must be used in all subsequent DMI commands.

Return value

```
DMIERR_NO_ERROR
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
DMIERR_CFE_RPC_NOT_SUPPORTED
DMIERR_RPC_INVALID_ADDRESS
DMIERR_RPC_TRANSPORT_UNAVAILABLE
DMIERR_RPC_SERVER_UNAVAILABLE
DMIERR_RPC_LOW_RESOURCES
```

DmiUnregister

```
DmiErrorStatus_t DMI_API DmiUnregister (
    /* [in] */  DmiHandle_t       handle
);
```

Description

DmiUnregister is an alias of *DmiRemoteUnregister*.

Indication Server Functions

The DMI 2.0 specification defines the SP responsibility to send indications and events to all management applications that have subscribed with the SP for this purpose. The DMI 2.0 specification also describes the mechanism for indication subscription and event filtering. This chapter specifies the interface and the required steps a management application should take to receive indications.

An engine on the managing side that enables indication delivery is called an Indication Server. The client front-end software implements an indication server for each of the underlying RPCs. An application must first activate the server before it can receive indications. The *DmiSetIndicationCallbacks* and *DmiIndicationListen* functions inform the client front end of indication entry points and inform the indication server to listen for indications. The *DmiIndicationListenExt* function starts the indication server for selected RPC and transport protocols.

Next, the application must subscribe for indications and events. Usually, the application subscribes after the indication server has been started, using the *DmiIndicationListen function*, for the following reasons:

- If an application subscribes for indications before the indication server is started, then any indication occurring on the SP will not be delivered. The Service Provider will detect that the indication server is absent and eventually will remove the corresponding subscription and event filter rows.

- To subscribe for indications an application should supply a “Subscriber Addressing“ attribute in the subscription row. This address is an address of the application’s indication server which is unknown until the server is started. This address includes dynamic end-point information needed to differentiate between multiple indication servers running on the same local node.

The *DmiGetSubscriptionAddress* function should be used to retrieve the indication server address for a given RPC and transport.

Subscribing for indications requires an application to register with the Service Provider– a potential indication generator. After adding subscription/event filter rows, an indication consumer may unregister itself from the SP. The indication delivery mechanism and the *DmiRegister* session are independent.

The Local DMI RPC and RAP RPC do not require an application to subscribe for indications. The local DMI RPC delivers all indications/events that are generated/forwarded by the local SP to all management applications that have started an indication server with the *DmiIndicationListen* function.

RAP RPC delivers all indications/events that are generated/forwarded by the Service Layer (SL) to all management applications registered with this SL. The only currently supported RPC that requires subscription is DCE RPC. For more details, see the *RPC Specific Information* section.

The indication server shutdown process includes removing the event filter subscription rows and calling the *DmiStopIndicationListening* or *DmiStopIndicationListeningExt* function to clean up RPC resources. Event filter rows should be deleted before removing

corresponding subscription rows, otherwise the Service Provider will automatically delete all associated filter rows and the application will get an error when it tries to delete non-existing filters.

DmiIndicationFuncs structure

The *DmiIndicationFuncs* structure identifies indication callback functions (entry points) provided by the Management Application. The indication function prototypes are discussed in the "Management Application Provider API" of the *DMI 2.0 Specification*. If the application is not interested in a particular indication type, then it can pass a NULL value for that function's address.

```
typedef  struct  DmiIndicationFuncs {
     DmiDeliverEvent*                    deliverEventFunc;
     DmiComponentAdded*                  componentAddedFunc;
     DmiComponentDeleted*                componentDeletedFunc;
     DmiLanguageAdded*                   languageAddedFunc;
     DmiLanguageDeleted*                 languageDeletedFunc;
     DmiGroupAdded*                      groupAddedFunc;
     DmiGroupDeleted*                    groupDeletedFunc;
     DmiSubscriptionNotice*              subscriptionNoticeFunc;
} DmiIndicationFuncs_t;
```

DmiSetIndicationCallbacks

```
DmiErrorStatus_t DMI_API DmiSetIndicationCallbacks (
   /* [in]   */  DmiIndicationFuncs_t*   indicationFuncs,
   /* [out] */  DmiIndicationFuncs_t*  oldIndicationFuncs
);
```

Description

This function passes information on indication callback entry points from the application to the client front end. Entry points set by this function are per-process globals. The initial value of all callback addresses is NULL. This means that no indications will be delivered until the *DmiSetIndicationCallbacks* function call changes the settings.

This function is executed entirely by the client and does not send any command to the Service Provider.

Parameters

Input

indicationFuncs new set of indication callbacks. If this parameter is NULL, the function does not change the current callback setting.

Output

oldIndicationFuncs Pointer to the application-supplied buffer to return the previous set of indication callbacks. If this parameter is NULL, the function does not return the old entry points.

Return value

`DMIERR_NO_ERROR`

DmiIndicationListen

```
DmiErrorStatus_t DMI_API DmiIndicationListen (  void );
```

Description

This function tells the client front end to listen for incoming indications. In response to this call, the client front-end software starts indication servers in all supported RPCs. Incoming indications will be delivered from the Service Providers where the application has subscribed to the application's indication server. The indication server forwards these indications to the application through the callback entry points last set by the *DmiSetIndicationCallbacks* function.

To receive indications, an application should complete the following operations:

1. Specify the indication entry points in the application using the *DmiSetIndicationCallbacks* function.
2. Start the indication server to listen for indications, using the *DmiIndicationListen* function, and retrieve the indication server address, using the *DmiGetSubscriptionAddress* function.
3. Subscribe for indications in all the Service Providers from which the application wishes to receive indications (*DmiRemoteRegister*, *DmiAddRow*, *DmiUnregister*). Set the "Subscriber Addressing" field of the subscription and event filter rows to the indication server address.

The *DmiIndicationListen* function returns to the caller immediately after processing all the instructions associated with this function. This function is executed entirely by the client and does not send any command to the Service Provider.

An application calls the *DmiIndicationListenExt* function to start indication server for specific RPC and transport protocols.

Return value

```
DMIERR_NO_ERROR
DMIERR_OUT_OF_MEMORY
DMIERR_CFE_ALREADY_LISTENING
DMIERR_CFE_NO_RPC_CLIENTS_LOADED
DMIERR_RPC_LOW_RESOURCES
```

DmiStopIndicationListening

```
DmiErrorStatus_t DMI_API DmiStopIndicationListening (  void )
```

Description

This function tells the client front end to stop delivering incoming indications to the application. The client front-end software stops all the RPC indication servers when an application calls this function. Callbacks already executing are allowed to complete. To resume receiving indications, an application should call the *DmiIndicationListen* function again.

This function is executed entirely by the client and does not send any command to the Service Provider.

DmiIndicationListenExt

```
DmiErrorStatus_t DMI_API DmiIndicationListenExt (
   /* [in] */   DmiString_t              *rpcType,
   /* [in] */   DmiStringList_t          *protList
);
```

Description

This function tells the client front end to listen for incoming indications for given RPC and transports. In response to this call, the client front-end software starts the specified indication server. To receive indications, an application must start at least one indication server. An application can call *DmiIndicationListenExt* multiple times for different RPC types, but only one indication server can be running for each specific RPC.

Incoming indications will be delivered from the Service Providers where the application subscribed with the specified RPC and one of the transports. The indication server forwards these indications to the application through the callback entry points last set by the *DmiSetIndicationCallbacks* function.

The *DmiIndicationListenExt* function returns to the caller immediately after processing all the instructions associated with this function. This function is executed entirely by the client and does not send any command to the Service Provider.

Input Parameters

rpcType RPC type of the indication server.

protList List of the indication server's transport types for the specified RPC type.

Return value

```
DMIERR_NO_ERROR
DMIERR_OUT_OF_MEMORY
DMIERR_CFE_ALREADY_LISTENING
DMIERR_CFE_NO_RPC_CLIENTS_LOADED
DMIERR_RPC_LOW_RESOURCES
```

DmiStopIndicationListeningExt

```
DmiErrorStatus_t DMI_API DmiStopIndicationListeningExt (
   /* [in] */   DmiString_t  FAR         * rpcType);
```

Description

This function tells the client front end to stop delivering indications via the specified RPC. The client front-end software stops the RPC indication server for the specified RPC when an application calls this function. Executing callbacks are allowed to complete. To resume receiving indications, an application should call the *DmiIndicationListenExt* or *DmiIndicationListen* function again.

This function is executed entirely by the client and does not send any command to the Service Provider.

Input Parameters

rpcType RPC type of the indication server.

Return value

```
DMIERR_NO_ERROR
DMIERR_RPC_TRANSPORT_UNAVAILABLE
```

DmiGetSubscriptionAddress

```
DmiErrorStatus_t DMI_API DmiGetSubscriptionAddress (
   /* [in]    */  DmiString_t  * rpcType,
   /* [in]    */  DmiString_t   * transportType,
   /* [out]   */  DmiString_t  ** address );
```

Description

This function returns the address of the indication server associated with the application, a given RPC and transport. This function requires that a local indication server was already started.

Applications should use the *DmiIndicationListen* function to start the indication server before calling this function.

The address returned by this function includes dynamic end-point information needed to differentiate among multiple indication consumers on the same machine. To provide the Service Provider with this information, an application should set the “Subscriber Addressing” field of the subscription and event filter rows to the retrieved address when subscribing for indications and events.

Parameters

Input

rpcType RPC type of the indication server. The only RPC types supported by this function are “*dce.*” Both “*local*” and “*rap*” RPC types do not require subscription for indications. For more detail, see the *RPC Specific Information* section below.

transportType

Transports available to the indication server. The function *DmiListTransportTypes* returns the run time list of available transports (see the *RPC Information Functions* section below).

Output

address Pointer to a returned pointer to the indication server address string. The application is responsible for cleaning up the allocated memory. For example, the application can use the *DmiFreeString* function to clean up memory in a default memory model.

Return value

```
DMIERR_NO_ERROR
DMIERR_OUT_OF_MEMORY
DMIERR_CFE_FUNCTION_NOT_SUPPORTED
DMIERR_CFE_RPC_NOT_SUPPORTED
DMIERR_RPC_SERVER_UNAVAILABLE
DMIERR_RPC_TRANSPORT_UNAVAILABLE
DMIERR_RPC_LOW_RESOURCES
```

Memory Handling

When programming with a DMI client front end that provides an abstraction layer of the underlying RPC environment and the CI interface, memory management issues must be especially considered. Two pieces of code are involved in the process of memory allocation/freeing: **application** code and **client** front-end code. It must be determined who and how memory for parameters passed between these entities is allocated and released.

For input parameters or simple data types such as *DmiId_t* and *DmiHandle_t*, the caller entity allocates and frees the memory. In such a case an application and client are absolutely independent from each other in memory usage. For instance, applications exclusively own the memory for all parameters of the *DmiOriginateEvent* function, while clients allocate and free memory for the *DmiDeliverEvent* function. In these cases, the memory management mechanism used by the application could differ from those used by the client.

However, when the DMI and client front end functions return output data passed as pointer-to-pointer parameters, the memory management rules are not so clear. In this case, the client and application should coordinate their memory management models; memory allocated by one piece of code must be freed by another piece. Follow these rules when allocating memory for output pointer-to-pointer parameters:

- If a function successfully returns, the caller always frees memory pointed to by an output parameter; the callee allocates this memory.
- If a function returns a failure error status, the caller does not free output parameters and the callee is responsible for memory cleanup.

- All pointers must be either NULL pointers or references to separately allocated memory blocks. This means that the only method allowed for complex data allocation is node-by-node allocation. This method requires a distinct allocation call for each memory block referenced by a pointer.

- Memory occupied by the top-level pointer is exclusively owned by the callee.

Implementation of these rules requires a mechanism that allows the exchange of knowledge on memory models between the application and the client. For this purpose, the client front end provides a special object called a memory descriptor which defines the memory management model. The *DmiSetDefaultMemDsc* and *DmiGetDefaultMemDsc* functions define the common (default) memory descriptor (memory model) to be used for allocating output parameters. This implies that both the application and the client must use this memory model whenever two different pieces of code are involved in memory allocation/freeing. For example, the client allocates the *DmiComponentList* structure for the *DmiListComponents* function and the application frees this structure after processing the received data. On the other hand, for the *CiGetAttribute* function, the component instrumentation code allocates the *DmiAttributeData* and the client performs the necessary clean up.

Initially the default memory management model is a standard system *malloc/free* model set by the client. When an application uses this memory model as a default model, it should use the *DmiAlloc/DmiFree* functions with the default descriptor (NULL) for memory management. The *DmiGetSysMemDsc* function returns the actual memory descriptor of this system memory model if for some reason the application needs it.

Note: Do not use *malloc/free* instead of *DmiAlloc/DmiFree* in this case. The implementation of *malloc*/*free* could be different depending on whether the debug or the release version of these functions are used.

The client front end provides one additional memory management model, pool memory, which allows the user to associate allocated blocks in a group (pool). See the description of the pool memory model in the *Pool Memory Management Model* section.

Applications can decide to use private memory management functions (*my_alloc*(), *my_free*()) to allocate/free output parameters, instead of using the client-supplied memory descriptors. In this case, the application should create a memory descriptor that represents its private model, using the *DmiCreateMemDsc* function. The application calls the *DmiSetDefaultMemDsc* function to inform the client that this memory descriptor will serve as the default. In this case, the application can use its private *my_alloc*() and *my_free*() functions to manage common client-application memory.

DMI applications usually perform a lot of memory allocation/free work when creating or deleting DMI structures. To ease memory handling in this situation, the client front end provides specific *DmiNew<type>* and *DmiFree<type>* functions for most of the DMI types. These functions can be customized to fit a particular memory model. The memory descriptor passed as one of the input parameters determines which memory model should be used to allocate/deallocate memory blocks. For detailed information on these functions, see *Creating DMI structures* and *Deleting DMI Structures.*

Memory Management Descriptor

The memory management descriptor is a representation of the memory model in the client front end. It defines how memory blocks are allocated/deallocated and how allocation failure is handled. Other aspects of the memory model, such as the grouping of memory blocks in pools and buffers, are not addressed directly in the descriptor. To allow such extensions, the memory management descriptor contains a reference to a private model data which can be used by the model-specific interface. The memory management descriptor is a handle defined as

typedef void * DmiMemDsc_t;

Effectively this handle references the following structure defining the memory model:

```
typedef  struct  DmiMemMgmt {
   DmiNodeAlloc_t      * nodeAlloc;
   DmiNodeFree_t       * nodeFree;
   DmiAllocFail_t      * allocFail;
   void                * mgmtData;
} DmiMemMgmt_t;
```

where:

nodeAlloc is the memory allocator

nodeFree is the memory deallocator

allocFail
is the handler routine called when the allocator fails to allocate memory

mgmtData
points to private model-specific data.

If the *allocFail* pointer is NULL then no handler is called in case of failure.

DmiNodeAlloc_t

```
typedef  void *  DMI_API DmiNodeAlloc_t (
   /* [in] */   size_t                  size,
   /* [in] */   void                    * mgmtData,
   /* [out ]*/  DmiErrorStatus_t        *  status
);
```

Description

DmiNodeAlloc_t is the placeholder for the memory allocation function. This function allocates a <*size*> byte memory block and returns the pointer to this block.

Parameters

Input

size Memory block size in octets.

mgmtData Pointer to the memory model private data.

Output

status Pointer to an address in memory where the DmiErrorStatus_t code is returned.

Return value

This function returns the pointer to the allocated block, or NULL in case of failure.

DmiNodeFree_t

```
typedef  DmiErrorStatus_t  DMI_API DmiNodeFree_t (
   /* [in] */   void   * ptr,
   /* [in] */   void   * mgmtData
);
```

Description

DmiNodeFree_t is the placeholder for the memory deallocation function. This function releases the memory block allocated by the corresponding *DmiNodeAlloc_t* function and pointed to by the <*ptr*> parameter.

Input Parameters

ptr Memory block to be freed.

mgmtData Pointer to memory model's private data.

Return value

This function returns the DmiErrorStatus_t code.

DmiAllocFail_t

```
typedef  void  DMI_API DmiAllocFail_t (
   /* [in ]*/ DmiErrorStatus_t   status
);
```

Description

DmiAllocFail_t is the placeholder of the allocation failure handler. The *DmiAlloc* function (described below) calls this function if it fails to allocate the memory block using the memory model allocator *DmiNodeAlloc_t*.

Input Parameters

status Allocation error code returned by *DmiNodeAlloc_t*.

Memory Management Descriptor Operations

To hide the specific *DmiMemMgmt* structure, the client front end provides functions for creating /deleting the memory management descriptor and allocating/deallocating memory in the specified model.

DmiCreateMemDsc

```
DmiMemDsc_t  DMI_API DmiCreateMemDsc (
   /* [in] */ DmiNodeAlloc_t*            nodeAlloc,
   /* [in] */ DmiNodeFree_t *            nodeFree,
   /* [in] */ DmiAllocFail_t *           allocFail,
   /* [in] */ void *                     mgmtData ,
   /* [out]*/ DmiErrorStatus_t *         status
);
```

Description

This function creates a memory management descriptor and fills it in with the specified parameters.

Parameters

Input

nodeAlloc Memory allocation function.

nodeFree Memory deallocation function.

allocFail Memory allocation failure handler.

mgmtData
Pointer to the memory model private data.

Output

status NULL, or a pointer to an address in memory where the DmiErrorStatus_t code is returned. If the <*status*> pointer is NULL then the error status is not returned.

The possible error codes are:

```
DMIERR_NO_ERROR
DMIERR_OUT_OF_MEMORY
DMIERR_MEM_INIT_FAILURE
```

Return value

When successful, this function returns the memory model descriptor. Otherwise, it returns NULL.

DmiDestroyMemDsc

```
DmiErrorStatus_t DMI_API DmiDestroyMemDsc (
    /* [in] */ DmiMemDsc_t                memDsc
);
```

Description

This function releases the memory management descriptor specified in <*memDesc*>.

Input Parameters

memDsc Memory descriptor to be destroyed.

Return value

```
DMIERR_NO_ERROR
DMIERR_OUT_OF_MEMORY
DMIERR_MEM_INIT_FAILURE
```

DmiAlloc

```
void *  DMI_API DmiAlloc (
    /* [in] */    size_t                   size,
    /* [in] */    DmiMemDsc_t              memDsc,
    /* [out]*/    DmiErrorStatus_t *       status
);
```

Description

This function allocates a <*size*>-byte memory block in the memory model specified by <*memDesc*>. When successful, it returns a pointer to the allocated block. Otherwise it returns NULL and calls the failure handler defined in <*memDesc*>.

Parameters

Input

size Memory block size in octets.

MemDsc Memory model descriptor. If <*memDsc*> is NULL, the default memory model is used.

Output

status Pointer to an address in memory where the DmiErrorStatus_t code is returned or NULL. If the <*status*> pointer is NULL then the error status is not returned.

Return value

This function returns the pointer to the allocated block or NULL in case of failure.

DmiFree

```
DmiErrorStatus_t  DMI_API DmiFree (
   /* [in] */   void                    * ptr,
   /* [in] */   DmiMemDsc_t             memDsc
);
```

Description

This function releases a memory block allocated in the memory model specified by the <*memDesc*> parameter and pointed to by the <*ptr*> parameter.

Input Parameters

ptr Memory block to be freed. If the <*ptr*> parameter is NULL , no memory is freed.

MemDsc Memory model descriptor. If the <*memDsc*> parameter is NULL, the default memory model is used.

Return value

This function returns the DmiErrorStatus_t code.

Default Memory Management Model

The default memory descriptor specifies how the application and the client allocate and deallocate memory when the allocating code is different from the code freeing the memory (refer to the discussion at the beginning of this section). The *DmiSetDefaultMemDsc* and *DmiGetDefaultMemDsc* functions pass the default memory management descriptor between the application and the client.

Functions that require the memory management descriptor <*mem_desc*> as a parameter use the default memory model when the descriptor is set to NULL.

Initially, the default memory management model is a standard system *malloc/free* model set by the client. The function *DmiGetSysMemDsc* returns the memory descriptor of this system memory model.

DmiSetDefaultMemDsc

```
DmiErrorStatus_t DMI_API DmiSetDefaultMemDsc
(
   /* [in]      */ DmiMemDsc_t           memDsc
);
```

Description

This function sets the <*memDsc*> parameter to the default memory management descriptor in all subsequent interactions between the application and the client. The descriptor set by this function is a global per-process variable.

Be especially careful not to mix memory models in different threads, when using this function in multi-thread applications.

Initially, the standard system descriptor is set as the default.

Input Parameters

memDsc New default memory descriptor. If the <*memDsc*> parameter is NULL, the current setting remains unchanged.

Return value

```
DMIERR_NO_ERROR
DMIERR_MEM_INIT_FAILURE
```

DmiGetDefaultMemDsc

```
DmiMemDsc_t  DMI_API DmiGetDefaultMemDsc (
   /* [out]*/   DmiErrorStatus_t        *  status
);
```

Description

This function returns the last set default memory management descriptor. The descriptor returned by this function is a global per-process variable. Initially, the standard system descriptor is set as the default.

Output Parameters

status NULL, or a pointer to an address in memory where the DmiErrorStatus_t code is returned. If the <*status*> pointer is NULL, the error status is not returned. The possible error codes are:

```
DMIERR_NO_ERROR
DMIERR_MEM_INIT_FAILURE
```

Return value

When successful, this function returns the default memory model descriptor. Otherwise, it returns NULL.

DmiGetSysMemDsc

DmiMemDsc_t DMI_API DmiGetSysMemDsc (

/* [out]*/ DmiErrorStatus_t * status

);

Description

This function returns the memory descriptor representing standard system *malloc/free* model provided by the client front end. This memory descriptor serves as a default memory descriptor until the application changes the setting, using the *DmiSetDefaultMemDsc* function.

Do not use *malloc/free* instead of *DmiAlloc/DmiFree* in this case. The implementation of *malloc*/*free* could be different depending on whether the debug or the release version of these functions are used.

Output Parameters

status NULL, or a pointer to an address in memory where the DmiErrorStatus_t code is returned. If the <*status*> pointer is NULL, the error status is not returned. The possible error codes are:

```
DMIERR_NO_ERROR
```

Return value

This function returns the system memory model descriptor.

DmiGetClientMemDsc

```
DmiMemDsc_t  DMI_API DmiGetClientMemDsc (
   /* [out]*/ DmiErrorStatus_t *      status
);
```

Description

This function always returns the NULL descriptor. In other words, the client uses the default memory model descriptor for allocating/freeing output parameters. This function is provided only for compatibility with the previous versions of the client front end.

Output Parameters

status NULL, or a pointer to an address in memory where the DmiErrorStatus_t code is returned. The only possible error code is DMIERR_NO_ERROR.

Return value

This function always returns the NULL descriptor.

Helper Functions

RPC Information Functions

The client front end defines the following RPC information functions:

- DmiListRpcTypes
- DmiListTransportTypes

These functions are detailed in the following sections.

DmiListRpcTypes

```
DmiErrorStatus_t DMI_API DmiListRpcTypes (
   /* [out]   */  DmiStringList_t **rpcTypes
);
```

Description

This function lists all RPC types currently available through the client front end. The members of the list differ, depending on the system configuration and installation options. See the *RPC Types* section for more detail on RPC types.

The application is responsible for cleaning up allocated memory in the default memory model.

Output Parameters

rpcTypes Pointer to pointer to list of client-allocated RPC types.

Return value

```
DMIERR_NO_ERROR
DMIERR_CFE_NO_RPC_CLIENTS_LOADED
```

DmiListTransportTypes

```
DmiErrorStatus_t DMI_API DmiListTransportTypes (
   /* [in]     */  DmiString_t *          rpcType,
   /* [out]    */  DmiStringList_t **     transportTypes
);
```

Description

This function lists all transports currently available for the specified RPC type. The members of the list differ, depending on the system configuration and installation options. See RPC Specific Information for more detail on transports for specific RPC type.

The application is responsible for cleaning up allocated memory in the default memory model.

This function is not supported for the "*rap*" RPC type.

Input Parameters

rpcType RPC type name. The *DmiListRpcTypes* function returns the list of available RPC types.

Output Parameters

transportTypes Pointer to pointer to list of client-allocated transport types.

Return value

```
DMIERR_NO_ERROR
DMIERR_CFE_FUNCTION_NOT_SUPPORTED
DMIERR_CFE_RPC_NOT_SUPPORTED
```

Pool Memory Management Model

Pool memory management allows an application to associate allocated memory to *pools*. This makes it more convenient to group allocated memory. You can create pools, allocate memory and associate it to a specific pool, or free pool memory.

Pools can also be destroyed, in which case all allocated memory belonging to that pool is also released.

DmiCreatePoolMemDsc

```
DmiMemDsc_t  DMI_API DmiCreatePoolMemDsc(
   /* [in] */ DmiAllocFail_t *           allocFail,
   /* [out]*/ DmiErrorStatus_t *         status
);
```

Description

This function is used to create a pool descriptor. Subsequent calls to *DmiAlloc*() should use a pool descriptor to associate allocated memory with that pool. Similarly, the *DmiFree* function in this memory model will release memory blocks belonging to the pool.

When successful, the function returns the memory model descriptor. Otherwise, it returns NULL.

If the <*status*> pointer is not NULL, then the referenced value will contain the DMI error code on return. Otherwise, an error status is not returned.

Note: Multiple active pools can exist simultaneously.

DmiDestroyPoolMemDsc

```
DmiErrorStatus_t DMI_API DmiDestroyPoolMemDsc (
   /* [in] */ DmiMemDsc_t                memDsc
);
```

Description

This function releases the pool descriptor specified in <*memDesc*>, and all memory allocated in this pool. This function returns a DMI error code.

Creating DMI structures

Functions for creating DMI structures are prefixed with `DmiNew`. The `DmiNew` functions create data of the type specified in the function's appendix. For example, `DmiNewTimeStamp` creates a new, zero-initialized `DmiTimeStamp`.

The `DmiNew` functions create the DMI data type structures in a memory model specified by the ***`memDsc`*** parameter. The contents of the newly-created data depend on the data type and the function's input parameters.

If a function does not have default input parameters other than memDsc, then only zero-initialized, top-level DMI data type structures are created. For example, the

`DmiNewRowData` function creates a `DmiRowData` structure with all the members initialized to zero.

Functions that create DMI list data types (for example, `DmiAttributeList` and `DmiAttributeIds`) allocate a list of specified size. These functions allocate both top-level structures (`{size, list}`) and zero-initialized arrays of corresponding elements. Top-level structures contain the specified list size and a pointer to the newly-created list.

Some functions let you specify the entire contents of the new data structure. The `DmiNewString` and `DmiNewUnicodeString` functions create new *`DmiString`* for specified ANSI or UNICODE strings, correspondingly. `DmiString` size and contents are determined by the input string.

The `DmiNewOctetString` function creates a new *`DmiOctetString`* of specified size and content.

The `DmiNewNodeAddress` function creates a *DmiNodeAddress* structure with specified *`address`*, *`rpc`* and *`transport`* fields.

DmiNew input parameters

`memDsc` memory model descriptor. If *`memDsc`* is NULL, then the default memory model is used.

`size` number of elements in the *`data_type`* list .

`srcString`
Null-terminated ANSI string.

`srcUnicodeString`
Null-terminated UNICODE string. In UNICODE, a null character is 2 octets.

`srcOctetString`
Pointer to array of octets.

`SizeOctet`
Number of octets in the *`srcOctetString`*.

`PAddress`
NULL-terminated string that defines the *`address`* body of the *`DmiNodeAddress`*.

`PRpc` NULL-terminated string that defines the *`rpc`* body of the *`DmiNodeAddress`*.

`pTransport`
NULL-terminated string which defines the *`transport`* body of the *`DmiNodeAddress`*.

DmiNew output parameters

status Pointer to an address in memory where the *DmiErrorStatus_t* code is returned or NULL. When the *status* pointer is NULL, the error status is not returned. Possible error codes are:

DMIERR_NO_ERROR

DMIERR_OUT_OF_MEMORY

DmiNew return values

If successful, the DmiNew functions return a pointer to a newly-allocated DMI *data_type*. Otherwise, they return NULL.

DmiNewTimestamp

```
DmiTimestamp_t*  DmiNewTimestamp (
    DmiMemDsc_t         memDsc,
    DmiErrorStatus_t *  status);
```

DmiNewString

```
DmiString_t* DmiNewString (
    char*                                src ,
    DmiMemDsc_t                          memDsc,
    DmiErrorStatus_t *                   status);
```

DmiNewOctetString

```
DmiOctetString_t* DmiNewOctetString (
    size_t                               size,
    char*                                src ,
    DmiMemDsc_t                          memDsc,
    DmiErrorStatus_t *                   status);
```

DmiNewDataUnion

```
DmiDataUnion_t*  DmiNewDataUnion (
    DmiMemDsc_t         memDsc,
    DmiErrorStatus_t *  status);
```

DmiNewEnumInfo

```
DmiEnumInfo_t*  DmiNewEnumInfo (
    DmiMemDsc_t         memDsc,
    DmiErrorStatus_t *  status);
```

DmiNewAttributeInfo

```
DmiAttributeInfo_t*  DmiNewAttributeInfo (
    DmiMemDsc_t         memDsc,
    DmiErrorStatus_t *  status);
```

DmiNewAttributeData

```
DmiAttributeData_t*  DmiNewAttributeData (
    DmiMemDsc_t          memDsc,
    DmiErrorStatus_t *   status);
```

DmiNewGroupInfo

```
DmiGroupInfo_t* DmiNewGroupInfo(
    DmiMemDsc_t          memDsc,
    DmiErrorStatus_t *   status);
```

DmiNewComponentInfo

```
DmiComponentInfo_t*  DmiNewComponentInfo (
    DmiMemDsc_t          memDsc,
    DmiErrorStatus_t *   status);
```

DmiNewFileDataInfo

```
DmiFileDataInfo_t*  DmiNewFileDataInfo (
    DmiMemDsc_t          memDsc,
    DmiErrorStatus_t *   status);
```

DmiNewClassNameInfo

```
DmiClassNameInfo_t*  DmiNewClassNameInfo (
    DmiMemDsc_t          memDsc,
    DmiErrorStatus_t *   status);
```

DmiNewRowRequest

```
DmiRowRequest_t*  DmiNewRowRequest (
    DmiMemDsc_t          memDsc,
    DmiErrorStatus_t *   status);
```

DmiNewRowData

```
DmiRowData_t*  DmiNewRowData (
    DmiMemDsc_t          memDsc,
    DmiErrorStatus_t *   status);
```

DmiNewAttributeIds

```
DmiAttributeIds_t*  DmiNewAttributeIds (
    size_t               size,
    DmiMemDsc_t          memDsc,
    DmiErrorStatus_t *   status);
```

DmiNewAttributeValues

```
DmiAttributeValues_t*  DmiNewAttributeValues (
    size_t               size,
    DmiMemDsc_t          memDsc,
    DmiErrorStatus_t *   status);
```

DmiNewEnumList

```
DmiEnumList_t*  DmiNewEnumList (
    size_t              size,
    DmiMemDsc_t         memDsc,
    DmiErrorStatus_t *  status);
```

DmiNewAttributeList

```
DmiAttributeList_t*  DmiNewAttributeList (
    size_t              size,
    DmiMemDsc_t         memDsc,
    DmiErrorStatus_t *  status);
```

DmiNewGroupList

```
DmiGroupList_t*  DmiNewGroupList (
    size_t              size,
    DmiMemDsc_t         memDsc,
    DmiErrorStatus_t *  status);
```

DmiNewComponentList

```
DmiComponentList_t*  DmiNewComponentList (
    size_t              size,
    DmiMemDsc_t         memDsc,
    DmiErrorStatus_t *  status);
```

DmiNewFileDataList

```
DmiFileDataList_t*  DmiNewFileDataList (
    size_t              size,
    DmiMemDsc_t         memDsc,
    DmiErrorStatus_t *  status);
```

DmiNewClassNameList

```
DmiClassNameList_t*  DmiNewClassNameList (
    size_t              size,
    DmiMemDsc_t         memDsc,
    DmiErrorStatus_t *  status);
```

DmiNewStringList

```
DmiStringList_t*  DmiNewStringList (
    size_t              size,
    DmiMemDsc_t         memDsc,
    DmiErrorStatus_t *  status);
```

DmiNewFileTypeList

```
DmiFileTypeList_t*  DmiNewFileTypeList (
    size_t              size,
    DmiMemDsc_t         memDsc,
    DmiErrorStatus_t *  status);
```

DmiNewMultiRowRequest

```
DmiMultiRowRequest_t*  DmiNewMultiRowRequest (
   size_t              size,
   DmiMemDsc_t         memDsc,
   DmiErrorStatus_t *  status);
```

DmiNewMultiRowData

```
DmiMultiRowData_t*  DmiNewMultiRowData (
   size_t              size,
   DmiMemDsc_t         memDsc,
   DmiErrorStatus_t *  status);
```

DmiNewString

```
DmiString_t FAR  * DmiNewString (
   const char FAR  * srcString ,
   DmiMemDsc_t   memDsc,
   DmiErrorStatus_t FAR *  status);
```

DmiNewUnicodeString

```
 DmiString_t FAR * DmiNewUnicodeString (
   const wchar_t FAR *  srcUnicodeString,
   DmiMemDsc_t memDsc,
   DmiErrorStatus_t  FAR *  status);
```

DmiNewOctetString

```
DmiOctetString_t FAR * DmiNewOctetString (
   size_t  sizeOctet,
   const char FAR * srcOctetString,
   DmiMemDsc_t memDsc,
   DmiErrorStatus_t FAR *  status);
```

DmiNewNodeAddress

```
DmiNodeAddress_t * DmiNewNodeAddress (
   const char *                          pAddress,
   const char *                          pRpc,
   const char *                          pTransport,
   DmiMemDsc_t                           memDsc,
   DmiErrorStatus_t FAR *                status);
```

Deleting DMI Structures

All functions for deleting DMI structures are prefixed with `DmiFree`. The `DmiFree` functions free data of the type specified in the function's appendix. For example, `DmiFreeTimeStamp` frees a `DmiTimeStamp`.

The `DmiFree` functions free DMI data structures in a memory model specified by *`memDsc`*. The `DmiFree` functions release the DMI data's top-level structure, and all recursively-referenced memory blocks. All blocks must be previously allocated in the *`memDsc`* memory model.

DmiFree input parameters

ptr Pointer to the DMI data to release.

memDsc Memory model descriptor. If *memDsc* is NULL, then the default memory model is used.

DmiFree return values

The DmiFree functions return these values:

DMIERR_NO_ERROR

DMIERR_OUT_OF_MEMORY

DmiFreeTimestamp

```
DmiErrorStatus_t  DmiFreeTimestamp (
    DmiTimestamp_t*     ptr,
    DmiMemDsc_t         memDsc);
```

DmiFreeString

```
DmiErrorStatus_t  DmiFreeString (
    DmiString_t*        ptr,
    DmiMemDsc_t         memDsc);
```

DmiFreeOctetString

```
DmiErrorStatus_t  DmiFreeOctetString (
    DmiOctetString_t*   ptr,
    DmiMemDsc_t         memDsc);
```

DmiFreeDataUnion

```
DmiErrorStatus_t  DmiFreeDataUnion (
    DmiDataUnion_t*     ptr,
    DmiMemDsc_t         memDsc);
```

DmiFreeEnumInfo

```
DmiErrorStatus_t  DmiFreeEnumInfo (
    DmiEnumInfo_t*      ptr,
    DmiMemDsc_t         memDsc);
```

Description

Frees a DmiEnumInfo pointed by <*ptr*> in a memory model specified by <*memDsc*>. Recursively releases all referenced memory blocks.

DmiFreeAttributeInfo

```
DmiErrorStatus_t  DmiFreeAttributeInfo (
    DmiAttributeInfo_t*                 ptr,
    DmiMemDsc_t                         memDsc);
```

DmiFreeAttributeData

```
DmiErrorStatus_t  DmiFreeAttributeData (
    DmiAttributeData_t*                    ptr,
    DmiMemDsc_t                            memDsc);
```

DmiFreeGroupInfo

```
DmiErrorStatus_t  DmiFreeGroupInfo (
    DmiGroupInfo_t*     ptr,
    DmiMemDsc_t         memDsc);
```

DmiFreeComponentInfo

```
DmiErrorStatus_t  DmiFreeComponentInfo (
    DmiComponentInfo_t*                    ptr,
    DmiMemDsc_t                            memDsc);
```

DmiFreeFileDataInfo

```
DmiErrorStatus_t  DmiFreeFileDataInfo (
    DmiFileDataInfo_t*  ptr,
    DmiMemDsc_t         memDsc);
```

DmiFreeClassNameInfo

```
DmiErrorStatus_t  DmiFreeClassNameInfo (
    DmiClassNameInfo_t*                    ptr,
    DmiMemDsc_t                            memDsc);
```

DmiFreeRowRequest

```
DmiErrorStatus_t  DmiFreeRowRequest (
    DmiRowRequest_t*    ptr,
    DmiMemDsc_t         memDsc);
```

DmiFreeRowData

```
DmiErrorStatus_t  DmiFreeRowData (
    DmiRowData_t*       ptr,
    DmiMemDsc_t         memDsc);
```

DmiFreeAttributeIds

```
DmiErrorStatus_t  DmiFreeAttributeIds (
    DmiAttributeIds_t*  ptr,
    DmiMemDsc_t         memDsc);
```

DmiFreeAttributeValues

```
DmiErrorStatus_t  DmiFreeAttributeValues (
    DmiAttributeValues_t*                  ptr,
    DmiMemDsc_t                            memDsc);
```

DmiFreeEnumList

```
DmiErrorStatus_t  DmiFreeEnumList (
    DmiEnumList_t*      ptr,
    DmiMemDsc_t         memDsc);
```

DmiFreeAttributeList

```
DmiErrorStatus_t  DmiFreeAttributeList (
    DmiAttributeList_t*                 ptr,
    DmiMemDsc_t                         memDsc);
```

DmiFreeGroupList

```
DmiErrorStatus_t  DmiFreeGroupList (
    DmiGroupList_t*     ptr,
    DmiMemDsc_t         memDsc);
```

DmiFreeComponentList

```
DmiErrorStatus_t  DmiFreeComponentList (
    DmiComponentList_t*                 ptr,
    DmiMemDsc_t                         memDsc);
```

DmiFreeFileDataList

```
DmiErrorStatus_t  DmiFreeFileDataList (
    DmiFileDataList_t*  ptr,
    DmiMemDsc_t         memDsc);
```

DmiFreeClassNameList

```
DmiErrorStatus_t  DmiFreeClassNameList (
    DmiClassNameList_t*                 ptr,
    DmiMemDsc_t                         memDsc);
```

DmiFreeStringList

```
DmiErrorStatus_t  DmiFreeStringList (
    DmiStringList_t*    ptr,
    DmiMemDsc_t         memDsc);
```

DmiFreeFileTypeList

```
DmiErrorStatus_t  DmiFreeFileTypeList (
    DmiFileTypeList_t*  ptr,
    DmiMemDsc_t         memDsc);
```

DmiFreeMultiRowRequest

```
DmiErrorStatus_t  DmiFreeMultiRowRequest (
    DmiMultiRowRequest_t*               ptr,
    DmiMemDsc_t                         memDsc);
```

DmiFreeMultiRowData

```
DmiErrorStatus_t  DmiFreeMultiRowData (
    DmiMultiRowData_t*  ptr,
    DmiMemDsc_t         memDsc);
```

DmiFreeNodeAddress

```
DmiErrorStatus_t DmiFreeNodeAddress (
    DmiNodeAddress_t *  ptr,
    DmiMemDsc_t         memDsc);
```

Copying DMI Structures

All functions for copying DMI structures are prefixed with `DmiCopy`. The `DmiCopy` functions copy data of the type specified in the function's appendix. For example, `DmiCopyTimeStamp` copies a `DmiTimeStamp`.

The `DmiCopy` functions copy DMI data structures. The destination, top-level structure must be previously allocated. All referenced structures of the destination structure are recursively created and copied from the respective referenced source structures. All allocations are done in the memory model specified by *`memDsc`*.

DmiCopy input parameters

`src` Pointer to the source `Dmi`*`data_type`* structure.

`dest` pointer to the destination `Dmi`*`data_type`* structure. The destination top level structure must be previously allocated.

`memDsc` Memory model descriptor. If *`memDsc`* is NULL, then the default memory model is used.

DmiCopy return values

`DMIERR_NO_ERROR`

`DMIERR_OUT_OF_MEMORY`

DmiCopyTimestamp

```
DmiErrorStatus_t DmiCopyTimestamp (
    DmiTimestamp_t FAR *dest,
    DmiTimestamp_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiCopyString

```
DmiErrorStatus_t DmiCopyString (
    DmiString_t FAR *dest,
    DmiString_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiCopyOctetString

```
DmiErrorStatus_t DmiCopyOctetString (
    DmiOctetString_t FAR *dest,
    DmiOctetString_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiCopyDataUnion

```
DmiErrorStatus_t DmiCopyDataUnion(
    DmiDataUnion_t FAR *dest,
    DmiDataUnion_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiCopyEnumInfo

```
DmiErrorStatus_t DmiCopyEnumInfo (
    DmiEnumInfo_t FAR *dest,
    DmiEnumInfo_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiCopyAttributeInfo

```
DmiErrorStatus_t DmiCopyAttributeInfo(
    DmiAttributeInfo_t FAR *dest,
    DmiAttributeInfo_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiCopyAttributeData

```
DmiErrorStatus_t DmiCopyAttributeData(
    DmiAttributeData_t FAR *dest,
    DmiAttributeData_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiCopyGroupInfo

```
DmiErrorStatus_t DmiCopyGroupInfo(
    DmiGroupInfo_t FAR *dest,
    DmiGroupInfo_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiCopyComponentInfo

```
DmiErrorStatus_t DmiCopyComponentInfo(
    DmiComponentInfo_t FAR *dest,
    DmiComponentInfo_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiCopyFileDataInfo

```
DmiErrorStatus_t DmiCopyFileDataInfo(
    DmiFileDataInfo_t FAR *dest,
    DmiFileDataInfo_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiCopyClassNameInfo

```
DmiErrorStatus_t DmiCopyClassNameInfo(
    DmiClassNameInfo_t FAR *dest,
    DmiClassNameInfo_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiCopyRowRequest

```
DmiErrorStatus_t DmiCopyRowRequest(
    DmiRowRequest_t FAR *dest,
    DmiRowRequest_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiCopyRowData

```
DmiErrorStatus_t DmiCopyRowData(
    DmiRowData_t FAR *dest,
    DmiRowData_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiCopyAttributeIds

```
DmiErrorStatus_t DmiCopyAttributeIds(
    DmiAttributeIds_t FAR *dest,
    DmiAttributeIds_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiCopyAttributeValues

```
DmiErrorStatus_t DmiCopyAttributeValues(
    DmiAttributeValues_t FAR *dest,
    DmiAttributeValues_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiCopyEnumList

```
DmiErrorStatus_t DmiCopyEnumList(
    DmiEnumList_t FAR *dest,
    DmiEnumList_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiCopyAttributeList

```
DmiErrorStatus_t DmiCopyAttributeList(
    DmiAttributeList_t FAR *dest,
    DmiAttributeList_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiCopyGroupList

```
DmiErrorStatus_t DmiCopyGroupList(
    DmiGroupList_t FAR *dest,
    DmiGroupList_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiCopyComponentList

```
DmiErrorStatus_t DmiCopyComponentList(
    DmiComponentList_t FAR *dest,
    DmiComponentList_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiCopyFileDataList

```
DmiErrorStatus_t DmiCopyFileDataList(
    DmiFileDataList_t FAR *dest,
    DmiFileDataList_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiCopyClassNameList

```
DmiErrorStatus_t DmiCopyClassNameList(
    DmiClassNameList_t FAR *dest,
    DmiClassNameList_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiCopyStringList

```
DmiErrorStatus_t DmiCopyStringList(
    DmiStringList_t FAR *dest,
    DmiStringList_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiCopyFileTypeList

```
DmiErrorStatus_t DmiCopyFileTypeList(
    DmiFileTypeList_t FAR *dest,
    DmiFileTypeList_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiCopyMultiRowRequest

```
DmiErrorStatus_t DmiCopyMultiRowRequest(
    DmiMultiRowRequest_t FAR *dest,
    DmiMultiRowRequest_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiCopyMultiRowData

```
DmiErrorStatus_t DmiCopyMultiRowData(
    DmiMultiRowData_t FAR *dest,
    DmiMultiRowData_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiCopyNodeAddress

```
DmiErrorStatus_t DmiCopyNodeAddress(
    DmiNodeAddress_t FAR *dest,
    DmiNodeAddress_t FAR *src,
    DmiMemDsc_t memDsc);
```

Duplicating DMI Structures

All functions for duplicating DMI structures are prefixed with `DmiDup`. The `DmiDup` functions duplicate data of the type specified in the function's appendix. For example, `DmiDupTimeStamp` duplicates a `DmiTimeStamp`.

The `DmiDup` functions duplicate top-level DMI data structures, and all referenced structures of the destination structure are recursively created and copied from the respective source structures. All allocations are done in the memory model specified by *`memDsc`*.

DmiDup input parameters

`src` Pointer to the source `Dmi`*`data_type`* structure.

`dest` Pointer to the returned pointer to the allocated destination `Dmi`*`data_type`* structure.

`memDsc` Memory model descriptor. If *`memDsc`* is NULL, then the default memory model is used.

DmiDup return values

`DMIERR_NO_ERROR`

`DMIERR_OUT_OF_MEMORY`

DmiDupTimestamp

```
DmiErrorStatus_t DmiDupTimestamp (
    DmiTimestamp_t  FAR **dest,
    DmiTimestamp_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiDupString

```
DmiErrorStatus_t DmiDupString (
    DmiString_t FAR **dest,
    DmiString_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiDupOctetString

```
DmiErrorStatus_t DmiDupOctetString (
    DmiOctetString_t FAR **dest,
    DmiOctetString_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiDupDataUnion

```
DmiErrorStatus_t DmiDupDataUnion(
    DmiDataUnion_t FAR **dest,
    DmiDataUnion_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiDupEnumInfo

```
DmiErrorStatus_t DmiDupEnumInfo (
    DmiEnumInfo_t FAR **dest,
    DmiEnumInfo_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiDupAttributeInfo

```
DmiErrorStatus_t DmiDupAttributeInfo(
    DmiAttributeInfo_t FAR **dest,
    DmiAttributeInfo_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiDupAttributeData

```
DmiErrorStatus_t DmiDupAttributeData(
    DmiAttributeData_t FAR **dest,
    DmiAttributeData_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiDupGroupInfo

```
DmiErrorStatus_t DmiDupGroupInfo(
    DmiGroupInfo_t FAR **dest,
    DmiGroupInfo_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiDupComponentInfo

```
DmiErrorStatus_t DmiDupComponentInfo(
    DmiComponentInfo_t FAR **dest,
    DmiComponentInfo_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiDupFileDataInfo

```
DmiErrorStatus_t DmiDupFileDataInfo(
    DmiFileDataInfo_t FAR **dest,
    DmiFileDataInfo_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiDupClassNameInfo

```
DmiErrorStatus_t DmiDupClassNameInfo(
    DmiClassNameInfo_t FAR **dest,
    DmiClassNameInfo_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiDupRowRequest

```
DmiErrorStatus_t DmiDupRowRequest(
    DmiRowRequest_t FAR **dest,
    DmiRowRequest_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiDupRowData

```
DmiErrorStatus_t DmiDupRowData(
    DmiRowData_t FAR **dest,
    DmiRowData_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiDupAttributeIds

```
DmiErrorStatus_t DmiDupAttributeIds(
    DmiAttributeIds_t FAR **dest,
    DmiAttributeIds_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiDupAttributeValues

```
DmiErrorStatus_t DmiDupAttributeValues(
    DmiAttributeValues_t FAR **dest,
    DmiAttributeValues_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiDupEnumList

```
DmiErrorStatus_t DmiDupEnumList(
    DmiEnumList_t FAR **dest,
    DmiEnumList_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiDupAttributeList

```
DmiErrorStatus_t DmiDupAttributeList(
    DmiAttributeList_t FAR **dest,
    DmiAttributeList_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiDupGroupList

```
DmiErrorStatus_t DmiDupGroupList(
    DmiGroupList_t FAR **dest,
    DmiGroupList_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiDupComponentList

```
DmiErrorStatus_t DmiDupComponentList(
    DmiComponentList_t FAR **dest,
    DmiComponentList_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiDupFileDataList

```
DmiErrorStatus_t DmiDupFileDataList(
    DmiFileDataList_t FAR **dest,
    DmiFileDataList_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiDupClassNameList

```
DmiErrorStatus_t DmiDupClassNameList(
    DmiClassNameList_t FAR **dest,
    DmiClassNameList_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiDupStringList

```
DmiErrorStatus_t DmiDupStringList(
    DmiStringList_t FAR **dest,
    DmiStringList_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiDupFileTypeList

```
DmiErrorStatus_t DmiDupFileTypeList(
    DmiFileTypeList_t FAR **dest,
    DmiFileTypeList_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiDupMultiRowRequest

```
DmiErrorStatus_t DmiDupMultiRowRequest(
    DmiMultiRowRequest_t FAR **dest,
    DmiMultiRowRequest_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiDupMultiRowData

```
DmiErrorStatus_t DmiDupMultiRowData(
    DmiMultiRowData_t FAR **dest,
    DmiMultiRowData_t FAR *src,
    DmiMemDsc_t memDsc);
```

DmiDupNodeAddress

```
DmiErrorStatus_t DmiDupNodeAddress(
    DmiNodeAddress_t FAR **dest,
    DmiNodeAddress_t FAR *src,
    DmiMemDsc_t memDsc);
```

RPC Specific Information

RPC Types

The table below lists RPC types supported by the client front end for WIN32. Only valid values of the <*rpc*> member of the *DmiNodeAddress* structure are listed. The function *DmiListRpcTypes* provides run time information on RPC clients available through the client front end.

RPC type value	Description
dce	OSF DCE RPC
local	SP-specific local DMI RPC
rap	DMI 1.0 SL remote access

DCE RPC

The client front end for WIN32 uses the Microsoft RPC implementation of DCE RPC. This section provides some information on Microsoft RPC addressing. Refer to the Microsoft RPC documentation for more detailed information.

The table below lists transport types supported by Microsoft RPC. The *DmiListTransportTypes* function provides run time information on transports available trough the DCE client.

Transport type	Description
ncacn_ip_tcp	Connection-oriented TCP over IP
ncacn_nb_tcp	Connection-oriented TCP over NetBIOS
ncacn_nb_ipx	Connection-oriented IPX over NetBEUI
ncacn_nb_nb	Connection-oriented NetBIOS over NetBEUI
ncacn_np	Connection-oriented named pipes
ncacn_spx	Connection-oriented SPX
ncacn_dnet_nsp	Connection-oriented DECnet transport
ncadg_ip_udp	Datagram (connectionless) UDP over IP
ncadg_ipx	Datagram (connectionless) IPX
ncalrpc	Local (cross-process) procedure call

The format and content of the <*address*> member of the *DmiNodeAddress* structure depend on the specified transport. It consists of network address and optional endpoint specified in the following syntax:

NetworkAddress
or
NetworkAddress[Endpoint]

The table below lists network address formats for each transport:

Transport	Network address	Example
ncacn_dnet_nsp	Host name or area and node syntax	ko 4.120
ncacn_ip_tcp ncadg_ip_udp	Host name or common internet address notation	ko 128.10.2.30
ncacn_nb_nb, ncacn_nb_tcp ncacn_nb_ipx	Server name	marketing
ncacn_np	Server name preceded by four backslash characters[19]	\\\\marketing
ncacn_spx ncadg_ipx	Server name or network number and node number preceded by tilde character	marketing ~0000000108002B30612C
ncalrpc	None	

When the network address is not specified (NULL string), the *DmiNodeAddress* parameter refers to the local host. Do not specify a network address when using the *ncalrpc* transport.

Endpoint information is an optional part of the <*address*> string. It is used only to differentiate multiple DMI indication servers on the same node. The function *DmiGetSubscriptionAddress* returns an endpoint as the part of its indication server address. To prevent indication loss, management applications should subscribe for indications with this address when more then one management application are running on the same computer.

Local DMI RPC

The local DMI RPC provides DMI 2.0 access to the local SP. It is implemented as a proprietary SP protocol for inter-process communication. It is recommended that you use

[19] a single backslash character (\) is interpreted as an escape character in the <*address*> field. To specify a single literal backslash character, you must supply two backslash characters (\\).

the local RPC when Component Instrumentation accesses local SP with the MI interface. The *DmiNodeAddress* specification for local DMI RPC is *{“ ”, “local” , “ ”}*.

The local DMI RPC behaves in a non-standard way when delivering indications to the management applications. The local DMI RPC delivers all indications/events that are generated/forwarded by the local SP to all management applications on this computer that have an indication server started with the *DmiIndicationListen* function. There is no need for management applications to subscribe for indications originating from the local DMI RPC. The *DmiGetSubscriptionAddress* function returns a DMIERR_CFE_FUNCTION_NOT_SUPPORTED error for a “local” rpc type. There is no event filtering for this type of RPC.

RAP RPC

The RAP RPC provides a limited DMI 2.0 interface for accessing the remote SL 1.x via RAP protocol.

The table below lists transport types supported by RAP.

Transport type	Description
tcpip	Connection-oriented TCP over IP
ipx	Datagram (connectionless) IPX
localt	Local (cross-process) procedure call

The table below lists possible network address formats for each transport:

Transport	Network address	Example
tcpip	Host name or common internet address notation followed by the string “:8000”	ko:8000 128.10.2.30:8000
ipx	Server name or network number and node number followed by four zeros	marketing A010000108002B30612C0000
ncalrpc	None	

RAP RPC does not support the following MI and client front-end functions:

- *DmiAddRow*
- *DmiDeleteRow*
- *DmiAddGroup*
- *DmiDeleteGroup*
- *DmiAddLanguage*

- *DmiDeleteLanguage*
- *DmiListLanguages*
- *DmiListTransportTypes*
- *DmiGetSubscriptionAddress*

The RAP RPC behaves in a non-standard way when delivering indications to the management applications. RAP RPC delivers all indications/events that are generated/forwarded by SL to all management applications registered with this SL. There is no need for management application to subscribe for indications coming from RAP RPC. There is no event filtering for this type of RPC.

Client Front-end error codes

The error codes listed in the following tables are the client front-end extensions to the standard Service Provider error code set defined in the *DMI 2.0 Specification*. Unless the error code is specified for a particular function, apply the following rule: DMIERR_CFE_* codes are common to all the client front-end functions, DMIERR_MEM_* codes can be returned by memory management functions and DMIERR_RPC_* codes can appear in functions involving RPC communication.

General Client Front-End Errors

Symbol	Value	Description
DMIERR_CFE_NO_RPC_CLIENTS_LOADED	0x800	Client front end did not find any RPC provider
DMIERR_CFE_RPC_NOT_SUPPORTED	0x803	Specified RPC type is not supported
DMIERR_CFE_FUNCTION_NOT_SUPPORTED	0x808	Function is not supported for specified RPC
DMIERR_CFE_ALREADY_LISTENING	0x809	Indication server is already started
DMIERR_CFE_GENERAL_FAILURE	0x80B	Client front-end non-specific error

Memory Management Errors

Symbol	Value	Description
DMIERR_MEM_INVALID_PARAMETER	0x805	Invalid parameter passed to memory management function
DMIERR_MEM_INIT_FAILURE	0x806	Client memory management initialization failure
DMIERR_MEM_NO_BLOCK_IN_POOL	0x807	Specified block is not found in memory pool

RPC Errors

Symbol	Value	Description
DMIERR_RPC_COMM_FAILURE	0x500	Network communication problem
DMIERR_RPC_INVALID_ADDRESS	0x501	Illegal address
DMIERR_RPC_TRANSPORT_UNAVAILABLE	0x502	Illegal or unsupported transport
DMIERR_RPC_SERVER_UNAVAILABLE	0x503	DMI server or indication server is not started
DMIERR_RPC_LOW_RESOURCES	0x504	RPC is out of resources
DMIERR_RPC_ACCESS_DENIED	0x505	RPC security violation
DMIERR_RPC_GENERAL_FAILURE	0x506	RPC non-specific error

Client Front-end programming examples

Memory handling

The following fragment of code illustrates using system memory model as a default model in a management application:

```
/*********************************************************************/
void main()
{

     DmiErrorStatus_t        status;
     DmiHandle_t             miHandle; /* management handle */
     DmiClassNameList_t      * classList=NULL;


     /* Register with the local SP.
     Application owns the memory allocated for miHandle. Client just fills it in with the
     actual value.
     */
     status = DmiRegister (&miHandle);
     if (!CheckStatus(status,"DmiRegister")) return ;

     /* Get class names of the first component.
     Client allocates classList in the default memory model.
     Since application did not change the default model, it is a    system memory model
     provided by client.
     */
     status = DmiListClassNames(
                 miHandle,
                 0,          /* list all groups     */
                 1,          /* component Id */
                 &classList);/* class name list */
     if (!CheckStatus(status," DmiListClassNames ")) return ;
     /*...... Processing class name list ..... */
     /* Application must free class name list in the default memory model - NULL memory
     descriptor used for this purpose.
     */
     DmiFreeClassNameList(classList, NULL);

     /* Close MA session */
     DmiUnregister (miHandle);
     return ;
}

/*********************************************************************/
```

The following fragment of code illustrates using private memory model as a default model in a component instrumentation application:

```
/*********************************************************************/
void main()
{
     DmiErrorStatus_t status;
     DmiMemDsc_t   myMemDsc;
```

```
    /*Create my private memory descriptor myMemDsc and tell the client front end to use
    it as default. */
    myMemDsc = DmiCreateMemDsc( MyAlloc, MyFree, NULL, NULL, &status);
    if (!CheckStatus(status, "DmiCreateMemDsc ")) return ;
    status = DmiSetDefaultMemDsc (myMemDsc);
    if (!CheckStatus(status, " DmiSetDefaultMemDsc ")) return ;

    /* From this point, all application threads should use C malloc/free functions when
    working with common application-client memory because MyMalloc and MyFree are wrapper
    functions of C malloc/free. */

    /* Register CI interface here. Include CiGetAttribute() function in the access list.
    */
    /* ....... */

    /* Wait for termination signal. */
    /* ........ */

    /* Unregister CI interface. */
    /* ....... */

    /*Destroy my private memory descriptor myMemDsc. */
    DmiDestroyMemDsc(myMemDsc);
    return;
}

/* MyAlloc() is my allocator for myMemDsc memory descriptor. */
void  * DMI_API MyAlloc (
    /* [in] */ size_t size,
    /* [in] */ void * mgmtData, /* not used in my memory model */
    /* [out]*/ DmiErrorStatus_t * status
)

{
    DmiErrorStatus_t tmpStatus=DMIERR_NO_ERROR;
    void * lpData;

    if ((lpData=malloc(size))==NULL)
      tmpStatus=DMIERR_OUT_OF_MEMORY;

    if (status) *status=tmpStatus;
    return lpData;
}

/* MyFree() is my deallocator for myMemDsc memory descriptor */
DmiErrorStatus_t DMI_API MyFree (
    /* [in] */  void * ptr,
    /* [in] */  void * mgmtData /* not used in my memory model */
)
{
    if (ptr) free(ptr);
    return DMIERR_NO_ERROR;
}

DmiErrorStatus_t DMI_API
CiGetAttribute(
```

```
    DmiId_t cid ,
    DmiId_t gid ,
    DmiId_t aid ,
    DmiString_t* language,
    DmiAttributeValues_t* keyList,
    DmiAttributeData_t** data)
{
    DmiErrorStatus_t    status = DMIERR_NO_ERROR;
    DmiAttributeData_t * attrData;

    /* <data> is a pointer-to-pointer output parameter so we need to allocate
    DmiAttributeData structure in a default memory model to be freed by client properly.
    main() has been set MyAlloc/MyFree as defaults. Actually, these are C malloc/free.
    Let's use them.
    */
    attrData = (DmiAttributeData_t *) malloc(sizeof        (DmiAttributeData_t));
    if (attrData == NULL)
    {
                status = DMIERR_OUT_OF_MEMORY;
    }
    else
    {
                /* set attribute type and value */
                attrData->data.type = MIF_INTEGER;
                attrData->data.value.integer = 1;
    }
    if (status == DMIERR_NO_ERROR)
    {
                /* set attribute Id */
                attrData->id = aid;
                *data=attrData;
    }
    else
    {
                 /* Free attribute data in case of error */
                  if (attrData) free(attrData);
    }

    return status;
}


/*********************************************************************/
```

The following fragment of code illustrates using pool memory model in a management application:

```
/*********************************************************************/
void main()
{

    DmiErrorStatus_t status;
    DmiHandle_t             miHandle; /* management handle */
    DmiAttributeValues_t* keylist;
    DmiDataUnion_t *data;
    DmiMemDsc_t   poolMemDsc;
```

```
    /*Create pool memory descriptor poolMemDsc. */
    poolMemDsc = DmiCreatePoolMemDsc(NULL, &status);
    if (!CheckStatus(status, " DmiCreatePoolMemDsc ")) return ;

    /* Register with the local SP. */
    status = DmiRegister (&miHandle);
    if (!CheckStatus(status,"DmiRegister")) return ;

    /* Set attribute. Allocate input parameters in the pool. */
    data = DmiNewDataUnion (poolMemDsc, &status);
    data-> type = MIF_INTEGER;
    data->value.integer = 1;
    keylist = DmiNewAttributeValues (1, poolMemDsc, &status);
    keylist->list[0].id = 1;
    keylist->list[0].data.type = MIF_INTEGER;
    keylist->list[0].data.value.integer = 10;

    DmiSetAttribute (miHandle, 3, 3, 3, keylist, DMI_SET, value);

    /* Destroy pool memory descriptor. By doing so, an application cleans up all the
    memory allocated in the pool. This includes
    < keylist > and <data> allocated memory.
    */
    DmiDestroyMemDsc(poolMemDsc);

    /* Close MA session */
    DmiUnregister (miHandle);

    return ;
}

/*********************************************************************/
```

Indication Consumer

The following fragment of code represents the skeleton of the indication consumer program.

```
/*********************************************************************/
#define N_O 3
void main()
{

    DmiErrorStatus_t status;
    DmiHandle_t             miHandle; /* management handle */
    DmiMemDsc_t   poolMemDsc; /*pool memory descriptor */
    DmiString_t * serverAddress; /* indication server address */
    /*machines originating events */
    char *machines[N_O]={"machine1","machine2","machine3"};
    DmiRowData_t * subscriptionRows[N_O]; /* indication subscription rows */
    DmiRowData_t * filterRows[N_O]; /* event filter rows */
    int i;
    DmiIndicationFuncs_t indicationFuncs = {myEventHandler, NULL,NULL,NULL,
                NULL,NULL,NULL,NULL}; /* indication callbacks*/
```

```
/*Create pool memory descriptor poolMemDsc for input parameters. */
poolMemDsc = DmiCreatePoolMemDsc(NULL, &status);
if (!CheckStatus(status, " DmiCreatePoolMemDsc ")) return ;

/* Tell to the client that myEventHandler will process incoming events. */
status=DmiSetIndicationCallbacks(&indicationFuncs,NULL);
if (!CheckStatus(status, "DmiSetIndicationCallbacks ")) return ;

/* Start indication server */
status = DmiIndicationListen();

/* Get the address of the indication server. It is allocated by the client.*/
status = DmiGetSubscriptionAddress (“dce”, “ncacn_ip_tcp”, &serverAddress);
/* Subscribe for events. Loop through the all machines of interest */
for (I=0; i< N_O; i++)
{
            DmiNodeAddress_t node;
             /* Allocate remote node address */
            node = DmiNewNodeAddress(machines[i], “dce”, “ncacn_ip_tcp”,
            poolMemDsc , &status);

            /* Register with the remote SP */
            status = DmiRemoteRegister (&miHandle,node);
            if (!CheckStatus(status,"DmiRegister")) return ;

            /* Create subscription row */
            subscriptionRows[i] = DmiNewRowData(poolMemDsc, &status);
            subscriptionRows[i]->className=
            DmiNewString("DMTF|SP Indication Subscription|001",
            poolMemDsc, &status);
            subscriptionRows[i]->keyList =
            DmiNewAttributeValues( 4, poolMemDsc, &status);
            subscriptionRows[i]-> values =
            DmiNewAttributeValues( 3, poolMemDsc, &status);

            /* Fill in subscription row with the appropriate data.
            Use indication server address for the “Subscriber Addressing”
            attribute.
            */
            /* ........*/
            subscriptionRows[i]->keyList >list[2].data.value.string =
            serverAddress;
            /* ........*/

            /* Add subscription row before adding filter row.*/
            status = DmiAddRow (miHandle, subscriptionRows[i]);
            if (!CheckStatus(status," DmiAddRow ")) return ;

            /* Create event filter row. */
            filterRows[i] = DmiNewRowData(poolMemDsc, &status);
            filterRows[i]->className=
            DmiNewString("DMTF|SPFilterInformation|001",
            poolMemDsc, &status);
            filterRows[i]->keyList =
            DmiNewAttributeValues( 6, poolMemDsc, &status);
            filterRows[i]-> values =
```

```
                DmiNewAttributeValues( 1, poolMemDsc, &status);

                /* Fill in filter row with the appropriate data.
                Use indication server address for the “Subscriber Addressing”
                attribute.
                */
                /* ........*/
                filterRows[i]->keyList->list[2].data.value.string serverAddress;
                /* ........*/

                /* Add filter row. */
                status = DmiAddRow (miHandle, filterRows[i]);
                if (!CheckStatus(status," DmiAddRow ")) return ;

                /* Close MA session */
                DmiUnregister (miHandle);

}

    /* Wait for termination signal while myEventHandler processes delivered
    events. */
    /* ........ */

    /* Remove indication subscriptions */
    for (I=0; i< N_O; i++)
    {
                DmiNodeAddress_t node;
                 /* Allocate remote node address. */
                node = DmiNewNodeAddress(machines[i], “dce”, “ncacn_ip_tcp”,
                poolMemDsc , &status);

                /* Register with the remote SP. */
                status = DmiRemoteRegister (&miHandle,node);
                if (!CheckStatus(status,"DmiRegister")) return ;

                /* Remove filter row before removing subscription row.*/
                status = DmiDeleteRow (miHandle, filterRows[i]);
                if (!CheckStatus(status," DmiDeleteRow ")) return ;

                /* Remove subscription row. */
                status = DmiDeleteRow (miHandle, subscriptionRows[i]);
                if (!CheckStatus(status," DmiDeleteRow ")) return ;

                /* Close MA session */
                DmiUnregister (miHandle);

}


    /* Stop indication server */
    DmiStopIndicationListening();

    /* Free indication server address with the default
    memory descriptor (allocated by the client) */
    DmiFreeString (serverAddress, NULL);
```

```
     /* Destroy pool memory descriptor. By doing so, an application cleans up all
     the memory allocated in the pool for subscription purposes */
     DmiDestroyMemDsc(poolMemDsc);

     return;
}

DmiErrorStatus_t DMI_API myEventHandler(
   /* [in] */ DmiUnsigned_t handle,
   /* [in] */ DmiNodeAddress_t *sender,
   /* [in] */ DmiString_t *language,
   /* [in] */ DmiId_t compId,
   /* [in] */ DmiTimestamp_t *timestamp,
   /* [in] */ DmiMultiRowData_t *rowData
 )
{
     /* Process incoming events here */
     /* .... */
     return DMIERR_NO_ERROR;
}
/*****************************************************************************/
```

<This page is intentionally blank.>

Chapter 8 - SDK Tools and Utilities

Overview

This chapter briefly describes the development tools included in this SDK:

- The DMI 2.0 MIF Conformance Checker
- The DMI Component Test System
- The Intel Service Provider setup utility for OEMs
- The DMI Explorer
- The DMI2SNMP Mapper

This chapter does not document these tools; refer to their respective online Help for usage details.

The DMI 2.0 MIF Conformance Checker tool

The DMI 2.0 MIF Conformance Checker (COMPCHK2) is a development tool designed for the Windows 95 and Windows NT[20] operating systems, and is used to benchmark user-developed component implementations. This program is intended to make it easier and faster to verify that your components conform with standardized groups definitions and development requirements. *COMPCHK2 does not itself define the requirements nor the standards for compliance.*

COMPCHK2 can be used to confirm that your group definitions:

- Use valid MIF grammar.
- Comply with existing DMTF-approved Standard Groups definition guidelines.
- Comply with your own standardized private group definitions.

COMPCHK2 can check your group definitions by either examining your MIF text files or querying the Service Provider for the group information.

The DMI 2.0 MIF Conformance Checker executable program is COMPCHK2.EXE. The COMPCHK2 tool has three file list work areas across the top, which are positioned over a scrollable test results log.

Use the Candidate MIFs work area (see the figure below) to build a list of component names or MIF files whose groups will serve as candidates for conformance testing by COMPCHK2.

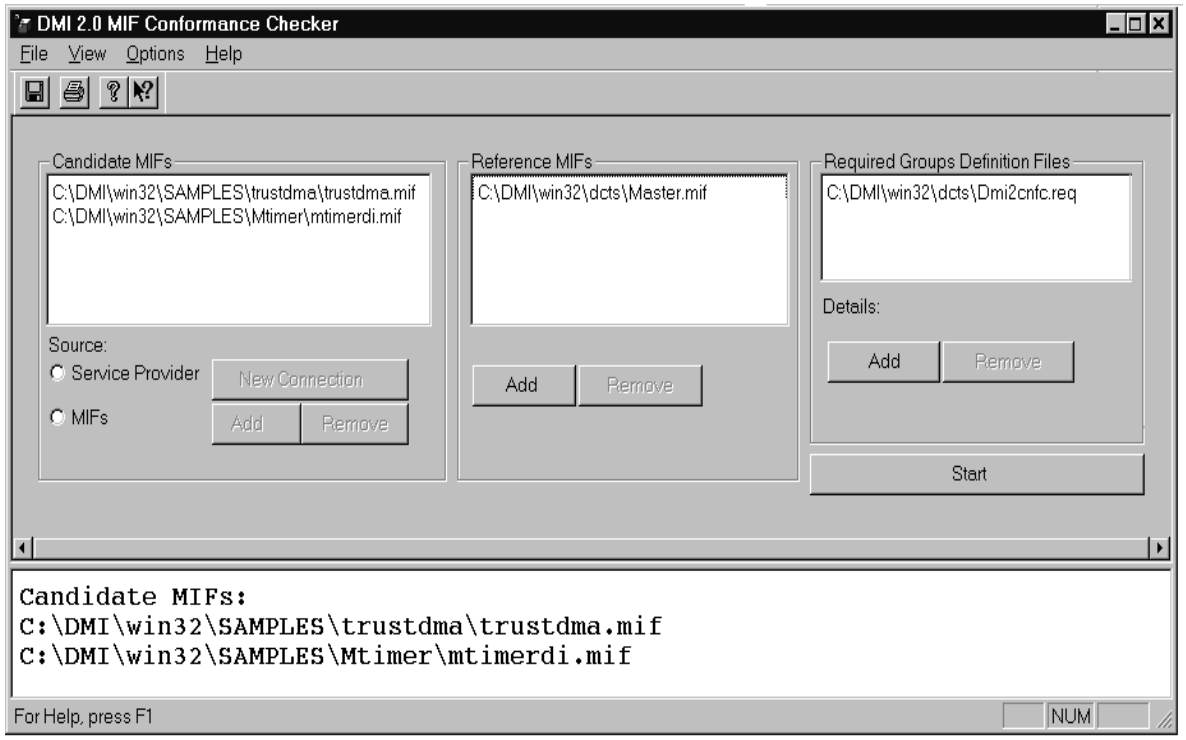
Use the Reference MIFs work area (see the figure below) to build a list of MIFs whose groups contain standardized group definitions. These standardized group definitions can be for either Standard Groups or your own private group definitions.

[20] Windows NT must be v3.51 or later.

One of the files included with COMPCHK2 is the MASTER.MIF file, which contains current[21] definitions of all DMTF-approved Standard Groups definitions and can be used to detect malformed Standard Groups definitions in the Candidate MIFs. The MASTER.MIF is updated as the standards change or new Standard Groups definitions are added. The latest DMTF-approved version of MASTER.MIF can be downloaded from the DMTF home page (see Chapter 1 for how to access this page).

Note: As a MIF is added to either list (Candidate or Reference), its MIF data are scanned and checked for conformance to the MIF grammar described in the *DMI 2.0 Specification*. If COMPCHK2 finds any errors or warnings to report, it creates a .ERR text file and displays it using the Notepad accessory provided with Windows.

Use the Required Groups Definition Files (REQ) work area to build a list of REQ files that describe groups that must be represented in the Candidate MIFs list.



[21] Current as of this release of COMPCHK2 only.

To check for group conformance, COMPCHK2 goes through the list of candidate groups in list order, comparing each candidate group class string to the class strings contained in the reference groups. When a candidate group class string matches a reference group class string, the groups corresponding to the class string (both candidate and reference) are compared. These standardized group fields *must* match:

- Class strings
- Group key counts and key attribute IDs
- Attribute counts[22] and IDs
- Attribute access methods
- Attribute enumeration counts and integer values
- Attribute data types

These groups fields are not standardized (are checked for syntactic but not semantic correctness) and *do not* have to match:

- Attribute values
- Maximum string length definitions
- Attribute and group names
- Attribute and group descriptions
- Attribute storage fields
- Groups IDs
- Enumeration string mappings

If the following groups are found, COMPCHK2 performs conformance checking as described below:

- Event generation group[23]—Any group definition that contains the class string "EventGeneration|<Specific name>|". It contains an associated group attribute, which is a class string. COMPCHK2 verifies that the group corresponding to the class string is contained in the component in which this event generation group is found.
- Event state group—Any group definition containing the class string "DMTF|Event State|". It contains a table of one or more rows of event generation groups. COMPCHK2 verifies that all event generation groups within this table are contained in the component in which this event state group is found.

A group definition may optionally include a pragma statement. When a pragma statement is found, COMPCHK2 performs conformance checking as described below:

---

[22] Within the Event Generation group, there is the concept of optional attributes (attributes found in the reference group that may or may not be found in the candidate group). Therefore, the actual attribute count of the candidate group may be different from the reference group; this would not cause the candidate group to fail. Refer to the document, *DMI 2.0 Baseline Conformance Specification*, for more information.

[23] See above footnote.

- Dependent_Groups keyword—If the reference group contains this keyword, the candidate group must also contain it. The groups listed here must be present in the component in which this `Dependent_Groups` pragma statement is found. All of the dependent groups found in the reference group's pragma must exist in the candidate group's pragma; the candidate group may also contain dependent groups, in addition to those found in the reference group.
- Implementation_Guideline keyword—If the reference group contains this keyword, the candidate keyword value must match. Possible values are: REQUIRED, OPTIONAL or OBSOLETE. The candidate cannot contain an implementation guideline pragma statement if the reference group does not.

To pass COMPCHK2's comparison to the reference groups, a candidate group must match all the standardized fields and comply with the conformance rules for the optional fields listed above. If it does not, the candidate group fails and the error is logged in the scrollable text frame within COMPCHK2's main window. In addition, any candidate group class strings that begin with the DMTF| prefix reserved for DMTF-approved Standard Groups definitions is assumed to be a Standard Groups type and *must* be found in the reference groups. If the candidate group is a Standard Groups type *but fails to match* any of the reference group class strings, the group fails and that error is logged.

To pass COMPCHK2's comparison to the Required Groups Definition Files list, all required group class strings within a given required groups definition file (REQ) must exist somewhere in the candidate system. However, *individual* candidate MIF files are not required to contain *every* group represented in a given REQ. If the composite list of all candidate groups lacks any one of the required groups within a given REQ, the candidate groups will have failed to conform to the requirements of that REQ, and this error will be logged in the scrollable text frame. One of the files included with COMPCHK2, DMI2CNFC.REQ, is a ready-to-use REQ file. The TEMPLATE.REQ is a tutorial document rather than a usable REQ file; it shows how REQ files are put together and how to describe requirements for MIF data definitions.

Note: Groups that appear in the Required Groups Definition Files list must be represented by group definitions in the Reference MIFs list.

Complete directions for using the COMPCHK tool can be found in its online Help documentation.[24]

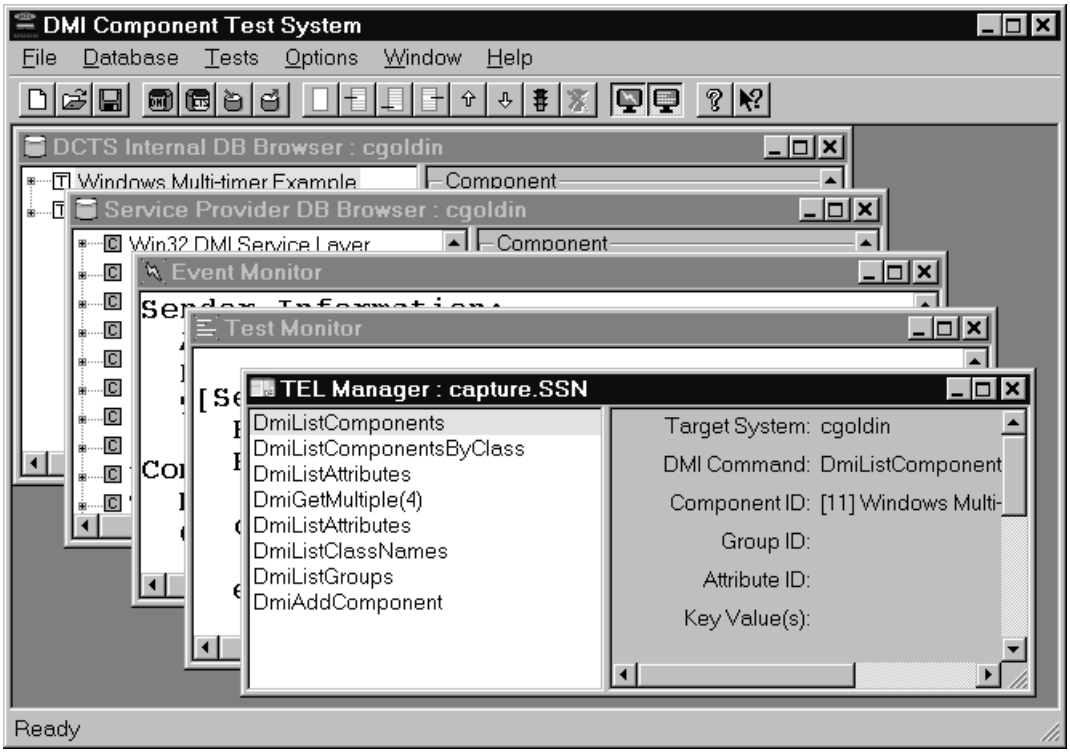
The DMI Component Test System tool

The DMI Component Test System (DCTS) executable program is DCTS2.EXE. DCTS is a development tool designed for Windows 95 and Windows NT[25] operating systems that allows the user to create, configure and run test suites that exercise DMI 2.0 Service Providers (both local and remotely connected via native RPC support), and any available DMI-enabled components and their instrumentation.

[24] See the RELNOTES.TXT file for known limitations in the Help system for this program.
[25] Windows NT must be v3.51 or later.

The DCTS2.EXE program provides a graphic user interface that directs you through the steps necessary to configure a test session and define its test list commands, browse components available for connected test systems, monitor events from the DMI-SP, and monitor test run data logging.

The DCTS user defines a connection list, which includes the test systems available and the RPC characteristics to use in connecting to them. The user also defines which component MIFs installed on a specified test system are to be testable by DCTS; a separate component database internal to DCTS keeps track of the test components for each connected system.



These test components are then available to the current Test Session and its test list, called a Test Execution List (TEL). The user defines the current TEL as a sequence of fully-defined DMI commands. Each TEL exercises DMI-SPs of the target connections, the test components being referenced there and relevant component instrumentation.

The exercise performed by a TEL can simulate actual runtime conditions by making repeated asynchronous calls of a command operating on a group or attribute, or by accessing several attributes, components and/or systems in rapid succession. Also, the user can configure the stress level—the test run modifiers—each time a TEL is submitted for execution by DCTS.

Complete directions for using the DCTS tool can be found in its online Help documentation.[26]

[26] See the RELNOTES.TXT file for known limitations in the Help system for this program.

Intel Service Provider setup utility for OEMs

The OEM setup utility is an InstallShield setup program which silently installs the DMI 2.0 Service Provider and required components on a Windows 95 and Windows NT system.

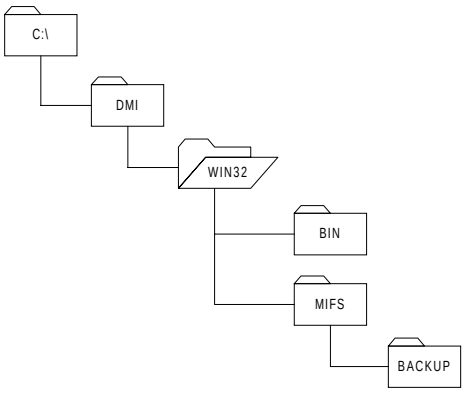
The OEM setup utility uses a response file as input for the silent installation program. You create the response file (see the *Creating a response file* section below), which replaces the user key strokes in the various dialog boxes used in a Windows-based developer installation. When you run the OEM setup utility using the response file, it does not have any user interface elements; all windows and dialog boxes during the setup process are invisible to the end user.

You can use the setup utility in conjunction with your own product setup program to install the Intel DMI 2.0 Service Provider on your customers' systems.

The InstallShield5 version 5.0.221 was used to create the OEM setup utility.

Components installed by the OEM setup utility

By default, the OEM setup program creates the UNINSTALL INTEL DMI 2.0 SP folder name and installs the following DMI 2.0 SP directories on the customer's system:



The setup program gets the installation information from the DMI_SP.ISS file. To change the base directory, C:\DMI\WIN32, you must edit the DMI_SP.ISS text file and change the "szPath" entry, or dynamically change the path through your main setup program.

If the DMI 2.0 Service Provider is not installed on the system, the OEM setup utility installs it. If an older version of the Service Provider is installed on the system, the setup utility replaces it. If a newer version of the DMI 2.0 Service Provider exists on the system, setup does not overwrite it.

Creating a response file

To use the setup utility, first run SETUP.EXE located in the OEMSETUP directory. Type this line:

```
setup.exe -r
```

This will create a response file called SETUP.ISS. Change the response file name to DMI_SP.ISS. You will use the response file to run the OEM setup utility and silently install the DMI 2.0 Service Provider on your customer's systems.

Using the OEM setup utility

To use the OEM setup program, perform the following steps:

1. Copy all the files in the \DMI\WIN32\UTILS\OEMSETUP directory to a subdirectory on the installation CD for your product.

2. From your setup script, invoke the SETUP.EXE program using the following option:

```
setup.exe -s -fdmi_sp.ins -f1dmi_sp.iss
```

The SETUP.RUL included in this SDK is an InstallShield example script file that shows how to call the DMI 2.0 Service Provider setup program from an InstallShield setup script. After the installation is completed, the setup program creates a file called DMI_SP.LOG in the SUPPORTDIR directory which contains the installation result. The SUPPORTDIR directory is a temporary directory used by InstallShield during installation. A value of "ResultCode = 0" in the log file indicates that the installation was successful.

You can copy and customize the source code of the *CallDmiSpSetupUtility()* function in the SETUP.RUL example script for your own product installation. To see how the above example setup program works, run the SETUP.EXE program in the \OEMSETUP subdirectory in this SDK. For more information on running silent installations and launching programs from InstallShield, refer to the InstallShield5 online reference manual.

Limitations

Since the OEM setup program does not restart the computer, your main setup program should instruct the user to do so for correct operation of the DMI 2.0 Service Provider.

Intel DMI Explorer

The Intel DMI Explorer is a development tool that can assist you in the development of management applications and in writing component instrumentation. The DMI Explorer enables you to view and modify DMI-compliant component information.

As a management application developer, you can use the DMI explorer to verify that the DMI information you expose in your application is accurate, by comparing it to the information displayed in the DMI Explorer.

As a component instrumentation writer, you can use the DMI Explorer to verify that the DMI-compliant component information is being transmitted accurately through the MI, to the management application. In other words, the DMI Explorer serves as your management application during the development process, until a management application is actually available.

Invoking Intel DMI Explorer

The DMI Explorer executable program is lddx.exe.

Type this command line to invoke Intel DMI Explorer:

```
lddx [/path path_string]  [/s]  [/name display_name]
```

where:

`/s` suppress splash screen.
(Optional)

`/name` change display name of the target computer to *`display_name`*. This name is shown in the tree pane root and in main frame title (default - remote address is used as machine name).
Note: If *`display_name`* contains spaces, be sure to use quotes ("). For example, `"Johns Machine"`.
(Optional)

`/path` use <*path_string*> as a path to the remote node. If not specified, DMI Explorer explores the local machine.
(Optional)

<*`path_string`*>
Full remote address, in this format: "`RPC|Transport|Address`"

where

`RPC` is RPC name (DCE).

`Transport`
is one of these pre-defined transport names: tcpip or spx.
or
is an RPC-dependent transport names supported by specified RPC (ncacn_ip_tcp, ncacn_spx, ...);

`Address`
is network address in RPC/transport-specific format.

Notes:

1. DCE can be accessed by SPX by either machine name or address. When using machine name, specify the path in this format: `DCE|SPX|machine_name`.
   When using machine address, specify the path in this format:
   `DCE|SPX|~machine_address`.
   Note: Be sure to add the symbol '~' before the address.

2. DCE TCP/IP can be accessed by either machine name or address. When using machine name, specify the path in this format: `DCE|tcpip|machine_name`.
   When using machine address, specify the path in this format:
   `DCE|tcpip|machine_address`.

Invocation Examples

This section presents some sample DMI Explorer invocation lines.

To connect to a machine named ilnt38, using the DCE TCP/IP protocol, type this line:

```
lddx.exe /path "DCE|tcpip|ilnt38"
```

To connect to a machine with the IP address 143.185.49.162, using the TCP/IP protocol, type this line:

```
lddx.exe /path "DCE|tcpip|143.185.49.162"
```

To connect to a machine named 0014013100aa0068a123, using the DCE SPX protocol, and displaying the machine name as "John," type this line:

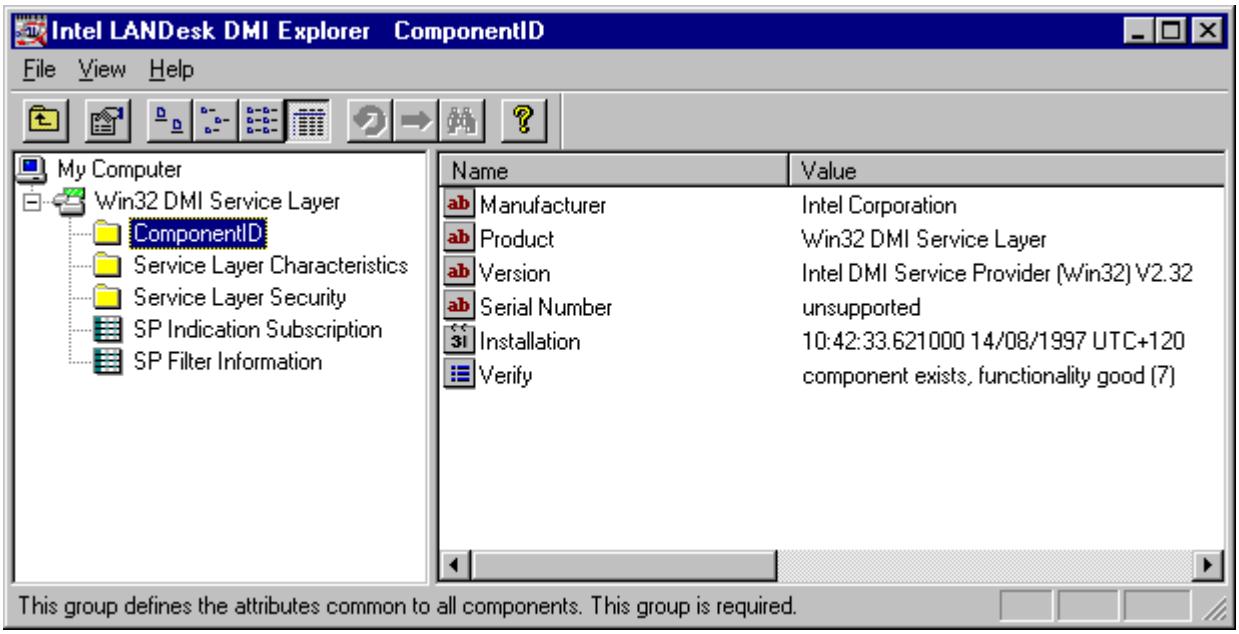
```
lddx.exe /path "DCE|SPX|~0014013100aa0068a123" /name John
```

To connect to a machine named ilnt38, using the `ncacn_ip_tcp` RPC transport, type this line:

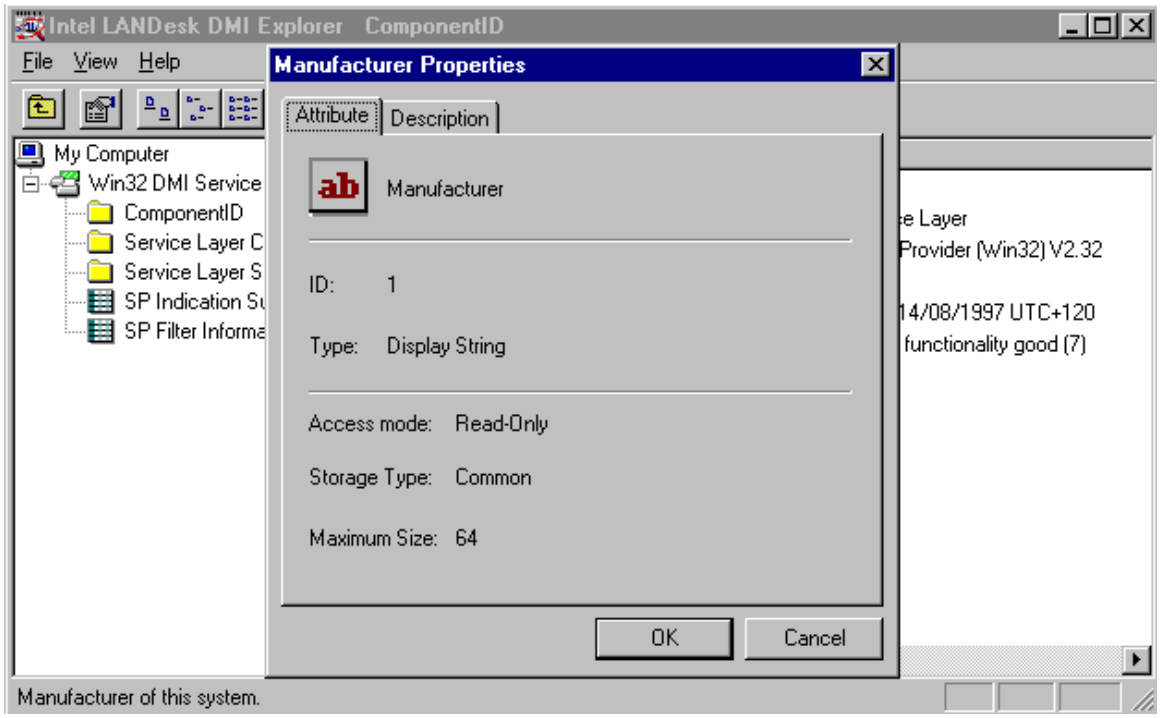
```
lddx.exe /path "DCE|ncacn_ip_tcp|ilnt38"
```

DMI Explorer Interface

For example, if you purchase a third-party component that is DMI-compliant and add the supplied MIF file (also referred to as component) into the MIF database, you can manage the component via the DMI Explorer. The DMI Explorer lists the components and component groups in the work area on the left (see the figure below). The information you can manage consists of attributes organized in groups and/or tables. Attributes are displayed in the work area on the right (see the figure below).



You can choose a component and see its associated attributes and attribute values. You can view additional information about an attribute on its property sheet (see figure below).



For Complete directions on using the DMI Explorer refer to its online Help.

DMI2SNMP Mapper

The Desktop Management Interface (DMI) is a standard management framework widely deployed to manage computer systems. The SNMP Management Framework (also known as the Internet-standard Network Management Framework) is widely used to manage network devices. The DMI and SNMP frameworks are similar in concept and function. However, the two are not inherently interoperable. Making the two protocols interoperable would greatly enhance enterprise management solutions, and provide an integrated network and system management capability.

The Desktop Management Task Force (DMTF), recognizing the need for leveraging both the ubiquity of SNMP-based server applications and the predominance of DMI-instrumented systems, is considering adopting a DMI-to-SNMP standard [27]. The Intel DMI2SNMP Mapper, which implements the DMTF standard, addresses the need to enable new SNMP applications that also manage DMI-instrumented systems.

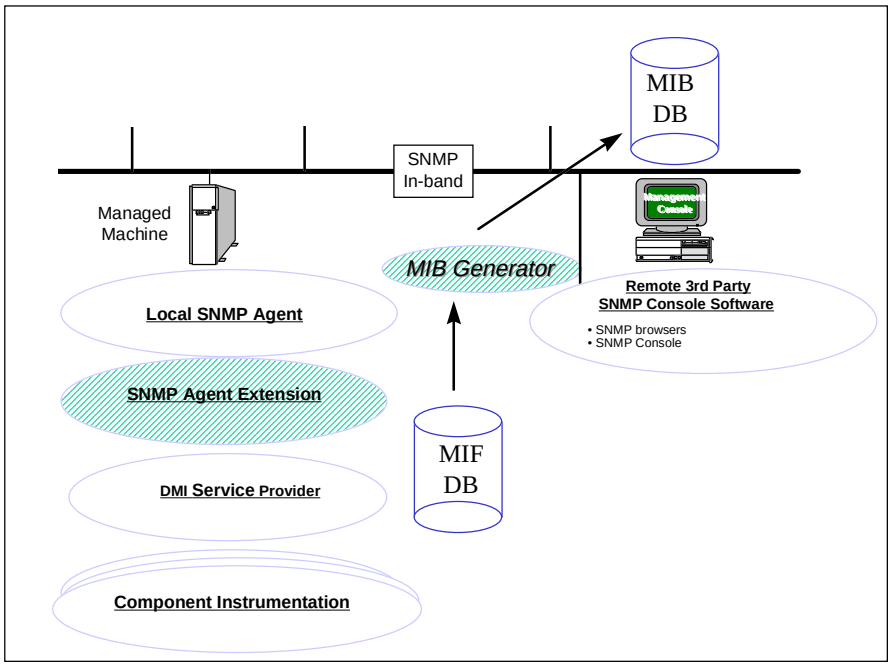
The DMI2SNMP Mapper extends the capabilities of SNMP-based management applications, so that they can manage DMI-instrumented systems. That is, the Mapper

[27] *DMTF SNMP to DMI Mapping Standard Candidate RFC 970710*, S. Bostock Novell, Inc. Brian O'Keefe Hewlett-Packard, Inc., July 10, 1997.

provides SNMP access to component MIF files installed on a DMI-instrumented system. The Mapper also facilitates writing a standard SNMP-based management application that can manage the functionality defined by a set of standard DMTF MIF groups, independent of both the component that implements those groups and of the system on which that component is installed.

DMI2SNMP Mapper Architecture

The DMI2SNMP Mapper provides the additional functionality to SNMP-based management in two components: MIF2MIB Generator and SNMP Agent Extension. The figure below illustrates the DMI2SNMP Mapper architecture.



The MIB Generator maps the DMI attributes in the MIF database into MIB objects, placing them in the MIB database located on the management console. The SNMP Agent Extension resides on DMI-enabled systems. It allows the systems to be managed by SNMP-based applications.

Files included in this release

The Intel DMI2SNMP Mapper includes:

- MIB Generator (mibgen.exe)
- SNMP Agent Extension (dmi2snmp.dll)
- Common DLL used by both the MIB Generator and the SNMP Agent Extension (dmtfoids.dll)
- Standard MIB files (dmtf-dmi.mib and master.mib)

Installing DMI2SNMP Mapper

Follow these steps to install the DMI2SNMP Mapper:

1. Create this registry key (using regedt32.exe): `SOFTWARE\\Intel\\DMI2SNMP`.
2. Create this registry key: `SOFTWARE\\Intel\\DMI2SNMP\Path`.
   Set its value to `C:\DMI\BIN\DMI2SNMP.DLL`.
3. Set the value of the first free string name in the `SYSTEM\\CurrentControlSet\\Services\\SNMP\\Parameters\\ExtensionAgents` registry key to `SOFTWARE\\Intel\\DMI2SNMP`.
4. Make the SNMP service dependent on win32sl.
5. Copy MIBGEN.EXE, DMI2SNMP.DLL, and DMTFOIDS.DLL to C:\DMI\BIN.

MIB Generator

The MIB generator (mibgen.exe) is a Windows*-based DMI 2.0 Management Application. Using the MIB Generator, you can select DMI components from a MIF database, and generate a MIB file containing the selected components. You can then install the newly-created MIB file on an SNMP Management Console.

Note: The MIB file generated by the MIB Generator references the DMTF-DMI MIB file. Therefore, the DMTF-DMI MIB file must be installed on the SNMP Management Console.

The MIB generator supports object ID Assignment and MIB Generation algorithms, as defined in RFC 970710, including TRAP-TYPE definitions. All groups are placed under the object ID (OID) node:

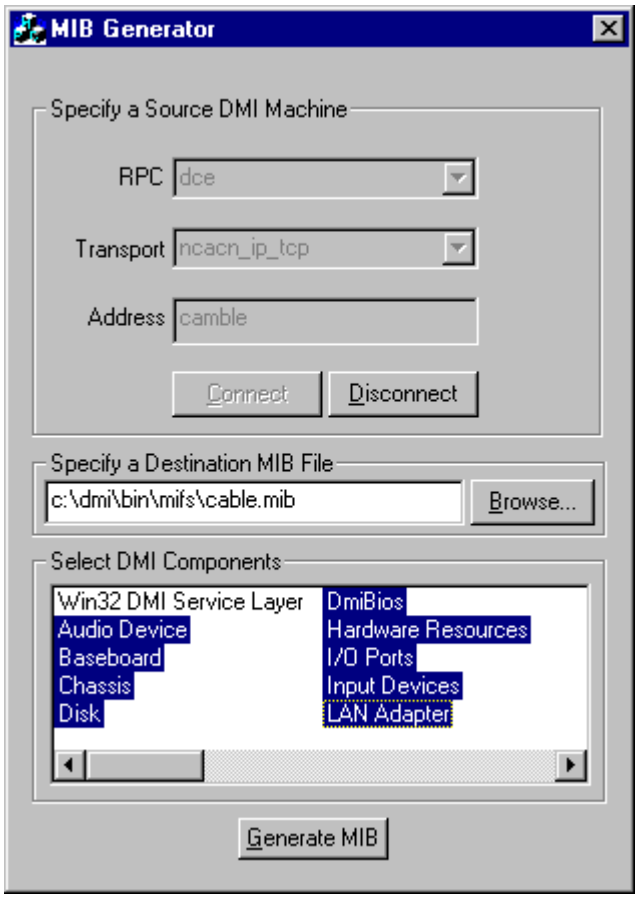
```
iso(1)org(3)dod(6)internet(1)private(4)enterprises(1)dmtf(412)
```

The DMTF Standard (RFC 970710) defines a MIB file (dmtf-dmi.mib) that describes the general DMI information (classes, components, etc.) This MIB file is an integral part of the mapping scheme defined in the DMTF standard; its implementation is mandatory for compliance with the DMI standard.

Using the MIB Generator to generate a MIB file

When you invoke the MIB Generator, it first connects to a DMI 2.0 Service Provider (locally or remotely). Follow these steps to create a MIB file, based on an existing MIF database (see figure below):

1. Specify the machine where the MIF file is located.
2. Specify a destination MIB file.
3. Select DMI components from an existing MIF database.
4. Generate the MIB file. This MIB file corresponds to all groups in the selected DMI components.



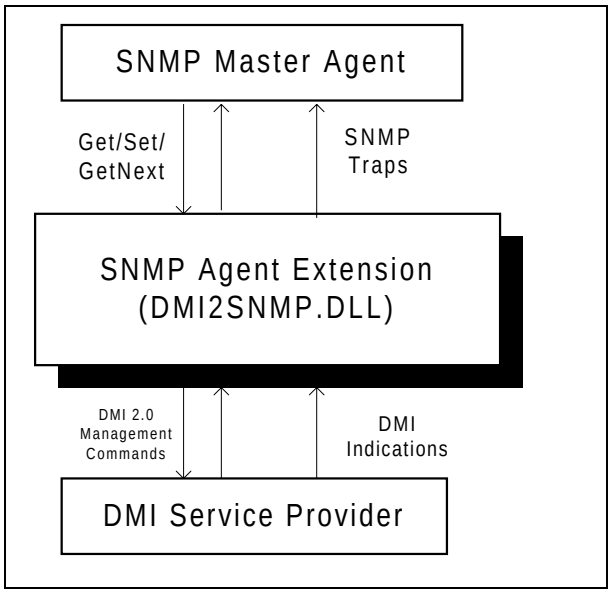
Note: If more than one group with the same "ClassName" string appears more than once in selected components, the MIB generator adds the corresponding object to the MIB file only once.

SNMP Agent Extension

The SNMP Agent Extension enables SNMP-based applications to manage a DMI-instrumented system, by mapping SNMP management requests to DMI commands.

The SNMP Agent Extension acts in two capacities (see figure below):

- a DMI 2.0 management application
- an extension to the Microsoft SNMP Agent



The SNMP Agent Extension is also implemented in accordance with RFC 970710. It supports requests to object IDs only under the node:

```
iso(1)org(3)dod(6)internet(1)private(4)enterprises(1)dmtf(412)
```

The SNMP Agent Extension does not support mapping based on the MIFTOMIB group, nor does it support "Intel|Error Control" style indications. It assumes that the current SDLINK.DLL (based on DMI 1.1) supports them.

MIB Files

The DMI2SNMP Mapper includes two reference MIB files: dmtf-dmi.mib and master.mib.

DMTF-DMI.MIB

The DMTF Standard (RFC 970710) defines a MIB file that describes the general DMI information (classes, components, etc.) This MIB file is an integral part of the mapping scheme defined in the DMTF standard; its implementation is mandatory for compliance with the DMI standard.

The MIB file generated by the MIB Generator references the DMTF-DMI MIB file. Therefore, DMTF-DMI.MIB must be installed on the SNMP Management Console.

MASTER.MIB

MASTER.MIB is the standard MIB file which corresponds to standard DMTF MASTER.MIF. It contains the definitions for all standard DMTF DMI groups.

APPENDIX A - Glossary of terms

Attribute	The building block of the Management Information Format (MIF); describes a single characteristic of a manageable product or component. A set of related attributes constitutes a group.
Class string	A text string that identifies a *group* outside the context of a particular *component* declaration. Identical group definitions will have identical class strings.
CMIP	Common Management Information Protocol, an OSI-based network management protocol standardized by ISO.
Command block	This term relates to the data block Management Interface documented in the *DMI 1.1 Specification.* This is the concatenation of data blocks (data structures) that constitute a command to be sent between *management applications* and the Service Provider, and between the Service Provider and *component instrumentation*.
COMPCHK2	DMI MIF Conformance Checker; a development tool. This tool is used to evaluate DMI v2.0 component MIF candidates by checking for correct syntax, conformance to user-defined requirements and conformance to the DMTF-approved Standard Groups definitions.
Component	Any hardware, software or firmware element contained in (or primarily attached to) a computer system.

Component instrumentation	A descriptive code or an ASCII file that defines manageability data about a PC, server or its hardware and software components. Instrumentation can cover everything from the computer's temperature to configuration data to what versions of software are installed. The data is specified in a standard format (see Management Information Format), so that manageability-sensitive operating systems and management applications can use it to prevent downtime and reduce the labor involved in system management. Instrumented systems can: • Show a complete inventory of all components. • Issue detailed alerts of pending failures. • Watch for indicators of a potential security breach. • Provide configuration and real-time or dynamic operational status to determine the "health" of the system.
Component Interface (CI)	The Component Interface (CI) is an API that handles communication between manageable elements and the DMI's Service Layer. The CI gives all hardware or software components (whether they're in or attached to a PC or server) a common method for describing their management attributes or features.
Confirm	This term relates to the data block Management Interface documented in the *DMI 1.1 Specification.* The final response from a *Request*.
Confirm buffer	This term relates to the data block Management Interface documented in the *DMI 1.1 Specification.* This is the area of memory where a *component instrumentation* or *Service Provider* puts response data.
DCTS2	DMI Component Test System; a development tool. This tool is used to build, organize and process test sessions that are used to exercise DMI v2.0 DMI-SPs, their components, and component instrumentation.
Direct interface	The method by which a *component instrumentation* informs the *Service Provider* that the instrumentation is already running. Rather than starting the code to service incoming requests, the Service Provider directs the requests to the code already running.

DMI	The industry's first set of OS-independent and protocol-independent application programming interfaces (APIs) for managing PCs, servers and other computer equipment. Developed by the Desktop Management Task Force (DMTF), the DMI provides manageability through three layers of software: Service Provider, Management Interface, and Component Interface.
DMTF	An industry consortium of more than 120 vendors, established in 1992 and committed to making PCs and servers easier to understand, use, configure and manage. Intel was a founding member of the DMTF and remains a leader in the organization. The DMTF developed the Desktop Management Interface and sponsors working groups that standardize manageability requirements for different classes of computer products.
Event	A runtime condition or change of state within a system.
Event Generator	A hardware or software device that has undergone a change in state or in which a certain condition of interest has occurred. This change of state or event directly or indirectly causes a new event to be processed by the Service Provider, which then produces and delivers an indication data structure to Event Consumers that have registered their interest in receiving indications.
Event Reporter	The software entity that causes a new DMI event to be processed by the Service Provider.
Event Consumer	A software entity that has registered with the Service Provider through the Management Interface with a non-null indication callback procedure address, i.e., it has registered to receive events.
Group	A collection of *attributes*. A group with multiple instances (rows) is called a *table*.
Indication	An unsolicited report resulting from an event, either from a *component instrumentation* to the *Service Provider*, or from the Service Provider to a *Management Application*.
ISO 8859-1	A character encoding standard defined by the International Standards Organization (ISO). Commonly known as extended ASCII or 8-bit ASCII.
Key	An identifier of a particular instance (row) of a table.
Keylist	A list of attribute IDs that is used as the index into a *table*.

Localized string	A version of a display string that is a translation of the original string into an equivalent string in the appropriate local language.
Management agent	A network management protocol agent (such as SNMP or CMOL) that can communicate to the DMI through the MI.
Management application (MA)	Code that uses the MI to request management activity from components.
Management Interface (MI)	The Management Interface (MI) is an API that provides the interface between the Service Layer and management applications and allows these applications to access, manage and control desktop systems and servers, components and peripherals. The MI offers a consistent interface for any management application to the various mechanisms used to obtain information from products and components within a desktop system or server. It also enables management applications to access information from any system or product regardless of vendor.
MIF	In the DMI architecture, an ASCII text file that describes a product's manageable features and attributes. The DMI maintains this information in a MIF database and makes it available to operating systems and management applications.
MIF database	The collection of known *MIF files*, stored by the *Service Provider* (in an implementation-specific format) for fast access.
MIF file	A file that uses the *MIF* to describe a *component*.
Octet	An 8-bit quantity.
Request	A function call or command block with associated context issued from the *Management Application* to accomplish a specific task.
Response	The data and status returned from a *request.*
Row	An instance of a *table*.
Service Provider (SP)	The Service Provider (SP) is a program that resides in the desktop system or server and is responsible for all DMI activities. This layer collects management information from products (whether system hardware, peripherals or software), stores that information in the DMI's database and passes it to management applications as requested.

SNMP	The most widely used protocol for communicating management information between the managed elements of a network and a network manager. SNMP focuses primarily on the network backbone; it is complemented by standards such as the DMI, which extend network manageability to the PC.
System	An IBM-PC or 100% compatible.
Table	A multidimensional *group*; a group with more than one instance (row).
Unicode	A character-encoding standard. Unicode characters are two octets each. When the first octet is zero, the second octet maps to the characters in ISO 8859-1.

<This page is intentionally blank.>

Index

E

F

G

H

I

K

M

O

P

R

S

T

U

V